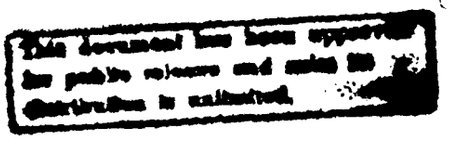
(4)

AD-A214 430

Transparency in Distributed File Systems

Richard Allen Floyd

Technical Report 272
January 1989



UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Transparency in Distributed File Systems

by

Richard Allen Floyd

Submitted in Partial Fulfillment
of the
Requirements for the Degree

Doctor of Philosophy

Supervised by Carla Schlatter Ellis

Department of Computer Science
College of Arts and Sciences

University of Rochester

Rochester, New York

1989

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1 REPORT NUMBER: 272 — 2 GOVT ACCESSION NO.	3 RECIPIENT'S CATALOG NUMBER
4 TITLE (and Subtitle): Transparency in Distributed File Systems	5 TYPE OF REPORT & PERIOD COVERED: Technical Report
	6 PERFORMING ORG. REPORT NUMBER
7 AUTHOR(s): Richard Allen Floyd	8 CONTRACT OR GRANT NUMBER(s): N00014-82-K-0193
9 PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Science Department, 734 Computer Studies Bldg., University of Rochester, Rochester, NY 14627	10 PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11 CONTROLLING OFFICE NAME AND ADDRESS: D. Adv. Res. Proj. Agency, 1400 Wilson Blvd., Arlington, VA 22209	12 REPORT DATE: January 1989
	13 NUMBER OF PAGES: 261
14 MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Res., Information Systems, Arlington, VA 22217	15 SECURITY CLASS. (of this report): Unclassified
	15a DECLASSIFICATION/DOWNGRADING SCHEDULE

16 DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17 DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18 SUPPLEMENTARY NOTES

None

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)

Heterogeneous Networks, File Reference Patterns, Directory Reference Patterns, File Migration

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)

The last few years have seen an explosion in the research and development of distributed file systems. Existing systems provide a limited degree of network transparency, with researchers generally arguing that full network transparency is unachievable. Attempts to understand and address these arguments have been limited by a lack of understanding of the range of possible solutions to transparency issues and a lack of knowledge of the ways in which file systems are used.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (Continued)

We address these problems by: 1) designing and implementing a prototype of a highly transparent distributed file system; 2) collecting and analyzing data on file and directory reference patterns; and 3) using these data to analyze the effectiveness of our design.

Our distributed file system, Roe, supports a substantially higher degree of transparency than earlier distributed file systems, and is able to do this in a heterogeneous environment. Roe appears to users to be a single, globally accessible file system providing highly available, consistent files. It provides a coherent framework for uniting techniques in the areas of naming, replication, consistency control, file and directory placement, and file and directory migration in a way that provides full network transparency. This transparency allows Roe to provide increased availability, automatic reconfiguration, effective use of resources, a simplified file system model, and important performance benefits.

Our data collection and analysis work provides detailed information on short-term file reference patterns in the UNIX environment. In addition to examining the overall request behavior, we break references down by the type of file, owner of file, and type of user. We find significant differences in reference patterns between the various classes that can be used as a basis for placement and migration algorithms. Our study also provides, for the first time, information on directory reference patterns in a hierarchical file system. The results provide striking evidence of the importance of name resolution overhead in UNIX environments.

Using our data collection analysis results, we examine the availability and performance of Roe. File open overhead proves to be an issue, but techniques exist for reducing its impact.

Curriculum Vitae

Richard Allen Floyd was born in Rochester, Minnesota on St. Patrick's Day, March 17th, 1953, a fact that his Irish mother has never let him forget. In 1971 he entered Iowa State University. He worked his way through school at a number of jobs, the most memorable of which was as a programmer for the High Energy Physics group at Iowa State. It was here that he was first introduced to computers and where he developed his distinctive late-night working style. He majored in Physics (with a minor in Electrical Engineering) and graduated, with distinction and several achievement awards, in 1977.

He started his graduate career at the University of California, Berkeley in September of 1977, but quickly discovered that it was the computer aspects of his earlier physics work that had made it so appealing. He left in 1978 to take a position at the Clinton P. Anderson Meson Physics Facility in Los Alamos, New Mexico, where he developed data acquisition and analysis software for particle physics experiments. He also began his education in Computer Science at the University of New Mexico. In 1981 he entered the University of Rochester's Computer Science program, and in 1982 was awarded a Masters in Computer Science. While at Rochester he was a teaching assistant and a research assistant.

Acknowledgements

My advisor, Carla Ellis, has put up with me for an extraordinary amount of time. In addition to being my advisor, she has been a good friend and colleague. Her help has meant more than I can say. I would not have been able to write this dissertation without her advice, support, and friendship. I would also like to thank the other past and present members of my committee, Tom LeBlanc, Chris Brown, Bezalel Gavish, Michael Scott, and Bruce Arden. Their direction and encouragement has been of great value.

The efforts of the Computer Science Department's faculty, students, and staff have made the department's facilities first rate. I would like to acknowledge in particular Liud Bukys, who provided extensive help and advice on various arcane aspects of RIG, and on IPC matters. He also developed a file access protocol that provided a starting point for the Roe file access protocol. Mike Dean ported the initial Local File Server to UNIX, Josh Tenenberg ported it to the Alto, and Doug Ierardi implemented the first Roe Transaction coordinator. In addition, Jill Forster and Rose Peet provided assistance with administrative details and endless encouragement.

Many friends, both in and out of the department, have helped with their encouragement and distractions. It's impossible to mention them all. I'd like to give special thanks to Kok Heong, for her boundless enthusiasm and her proofreading, to Jaii, who supported my fondness for things New Mexican, to E, for too many things to list, and to Mark, Brux, Jennifer, Liralen, George, Judy, Lee, Joan, Mike, Beth, Jim, Stuart, Diane, Steve, and Lydia.

This work has been supported in part by the National Science Foundation under grant number DCR-8320136, in part by the Office of Naval Research under grant number N00014-82-K-0193, and, thanks to the efforts and faith of Steve Vinter, in part by BBN Laboratories.

Abstract

The last few years have seen an explosion in the research and development of distributed file systems. Existing systems provide a limited degree of network transparency, with researchers generally arguing that full network transparency is unachievable. Attempts to understand and address these arguments have been limited by a lack of understanding of the range of possible solutions to transparency issues and a lack of knowledge of the ways in which file systems are used.

We address these problems by: 1) designing and implementing a prototype of a highly transparent distributed file system; 2) collecting and analyzing data on file and directory reference patterns; and 3) using these data to analyze the effectiveness of our design.

Our distributed file system, Roe, supports a substantially higher degree of transparency than earlier distributed file systems, and is able to do this in a heterogeneous environment. Roe appears to users to be a single, globally accessible file system providing highly available, consistent files. It provides a coherent framework for uniting techniques in the areas of naming, replication, consistency control, file and directory placement, and file and directory migration in a way that provides full network transparency. This transparency allows Roe to provide increased availability, automatic reconfiguration, effective use of resources, a simplified file system model, and important performance benefits.

Our data collection and analysis work provides detailed information on short-term file reference patterns in the UNIX environment. In addition to examining the overall request behavior, we break references down by the type of file, owner of file, and type of user. We find significant

differences in reference patterns between the various classes that can be used as a basis for placement and migration algorithms. Our study also provides, for the first time, information on directory reference patterns in a hierarchical file system. The results provide striking evidence of the importance of name resolution overhead in UNIX environments.

Using our data collection analysis results, we examine the availability and performance of Roe. File open overhead proves to be an issue, but techniques exist for reducing its impact.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1. The Problem

Recent years have seen a dramatic decrease in both the cost and size of computer systems. This has encouraged the proliferation of *local area networks (LANs) of small machines*. With this trend has come the problem of sharing information among users of these machines. The problem of file sharing, in particular, is one that all such networks must address.

It is not enough to provide users with the means to specify and access files on remote machines. The complexity introduced by the distribution of computing resources and files will quickly overwhelm users unless we provide a mechanism that allows simple abstractions of the environment to be constructed. In addition, this mechanism should allow users to make effective use of the resources available, and to take advantage of the distributed nature of the environment. Our approach to achieving these goals, a *highly network transparent distributed file system*, is the subject of this dissertation.

The remainder of this chapter provides an introduction to the problem of sharing files in a LAN, to transparent distributed file systems, and to this dissertation. Section 1.2 maps out the range of solutions that can be used to address the problem and argues that a highly transparent distributed

file system is, in general, the preferred solution. Section 1.3 presents the major issues that arise in designing such a file system. Sections 1.4 and 1.5 describe the approach used in our work and summarize the rest of the dissertation. Section 1.6 explains the significance of our results.

1.2. A Range of Solutions

1.2.1. Utilities for File Transfer

Perhaps the simplest solution for dealing with file sharing in a network is to provide utilities that allow users to explicitly copy files from machine to machine. Two examples of file transfer utilities are the *ftp* command (used in the Internet environment [Postel 82]) and the *rcp* command (used between BSD UNIX[1] systems [Quarterman 85]).

There are a number of problems with this approach to file sharing. It requires that users know the hosts on which their files are located and that their applications run on these hosts. It makes it difficult to organize files, as they may be scattered across a number of hosts. Moving files (to avoid downtime, load balance, and so on) requires knowledge of the hosts and resources available on the network. In a heterogeneous environment, users see several different naming conventions and file transfer protocols. Given the ability to transfer a file, users frequently react by spreading copies throughout the network to increase the file's availability. This raises the difficult problem of keeping these copies consistent or, if they do become inconsistent, the problems of locating the most current one and reconciling conflicting changes.

1.2.2. Transparent Remote Access

A more sophisticated solution to the problem of file sharing in a network is to provide *transparent access* to files on remote machines. By *transparent access* we mean that files on remote machines are accessed using the same techniques used for local files. One approach for achieving this is to allow a host or other device name to be explicitly included in a file specification, and to have

[1] UNIX is a trademark of AT&T Bell Laboratories.

lower layers route the call to the appropriate host. VAX/VMS and RIG [Ball 76] support this approach.

An alternative commonly used in the UNIX environment is to provide a way to patch a remote naming tree into the local naming tree. This allows remote files to be accessed by users without necessarily being aware of the existence of these remote hosts. This approach is used by Sun's NFS [Lyon 85] and AT&T's RFS [Rifkin 86].

The major advantage of transparent access is that it can, with a careful choice of naming conventions, allow a user to run applications and access files independent of relative user and file locations. In the case of systems (such as NFS and RFS) that make the remote naming tree an extension of the local one, it allows users to take advantage of the file resources of a network without generally needing to know the details of their location.

There are a number of serious drawbacks to this approach, though. Binding sections of the naming tree to a host means that the resources available to users of that tree are limited to those on that machine. Limitations in storage space and processor capacity crop up in unexpected places, and reconfiguration to correct problems is difficult. A more serious drawback is decreased availability due to machine crashes and other faults. The availability of an NFS-style system will most likely be less than that of a host running alone, as users will generally require access to multiple machines to perform their work. In addition, users lose the ability to replicate important files on a number of different machines. The ability to replicate critical resources to improve availability is a key advantage of networks over single site systems. A solution that takes advantage of the potential benefits of distribution is needed.

1.2.3. Transparent Distributed File Systems

A solution to the problems described in the last two sections is to provide users with a network transparent distributed file system. By *network transparent* we mean that the distributed file system (DFS) allows users to create and access objects with no constraints due to the name, the location of the user and object, and with no knowledge of the underlying systems. Our basic goal is

to preserve the abstraction of a single, shared file system across the network.

A major benefit of this approach is that it frees users from the need to understand the details of the underlying network, and so allows them to develop a simpler model of their environment. Uncoupling name from location allows the underlying system to be adapted to changing demands by transparently adding or removing storage and to improve performance by adjusting the locations of files to match requests. Further, this uncoupling allows files to be replicated and distributed to enhance availability.

1.3. Issues in the Design of a Transparent Distributed File System

There are a number of issues that arise in the design of a distributed file system. For example, what is the model of the file system that will be presented to the user? If a decision is made to give the user the view that a single file system exists, how do we support this view? What actions are taken if failures or resource limitations make it difficult to maintain this view? Are there ways that we can minimize the occurrence of these problems? How do we support such a file system in a heterogeneous environment? What is the coupling between names and objects stored in the file system? How are objects located? What effect do architectural decisions have on availability and performance? If the DFS supports replication, do we insist on consistency? Are these decisions dependent on the characteristics of the underlying hosts and network?

The last few years have seen an explosion in the research and development of network transparent DFSs that address many of the issues we have listed here. These efforts have been hampered by two factors:

(1) Limited understanding of the range of possible solutions to problems in DFS design and of the interactions of these solutions.

(2) A lack of understanding of the ways in which file systems are used

There exist a wide range of solutions to various individual problems that a network transparent DFS must solve. Examples include numerous algorithms for naming, consistency control, file

placement and file migration. For the most part, however, these solutions exist in isolation. There is little knowledge of how they would interact in a fully transparent DFS. Distributed file systems that do exist have abandoned one or more of the aspects of transparency in order to simplify design. For example, LOCUS [Walker 83b] takes the approach of gluing together the name spaces of existing hosts (with additional mechanisms for limited replication), and so provides access transparency, but only limited location transparency. The IBIS file system [Tichy 84], on the other hand, supports its own name space and so can provide location transparency, but does not guarantee that users at different locations will see consistent versions of files.

Further, most existing systems have been designed for a limited environment. In particular, there has been little attempt by DFSs to accommodate heterogeneous environments. The rapid evolution of computer hardware and software, the proliferation of special-purpose machines, and the continued growth in the use of networks are all factors that encourage heterogeneity. DFSs that are designed for heterogeneous environments are faced with the problem of accommodating the differing capabilities and needs of the underlying systems.

Because of the modest transparency goals and environmental limitations of current systems, there is no real understanding of the interactions of solutions that can be used in a transparent DFS, and of the range of applicability of these solutions. Nor is it clear to what extent the various aspects of transparency are realizable.

Evaluating the effectiveness of a DFS design and implementation requires knowledge of the ways in which it will be used. There is, unfortunately, very little detailed information available on the usage patterns for a DFS, or even of single site file systems. In particular, most DFSs currently in existence use a hierarchical directory system modeled after that used in UNIX, but there are no data available on directory reference patterns.

1.4. The Thesis

Our thesis is as follows:

> Full network transparency in distributed file systems offers significant benefits. These benefits include increased availability, more effective use of resources, the ability to adapt to changing demands, transparent reconfiguration to adjust to changes in resources, a greatly simplified file system model from the user/application point of view and, with careful design, enhanced performance over other DFS approaches.

Distributed file system designers recognize that there are desirable aspects of network transparency, but have generally argued that heterogeneity, complexity, performance, availability, and autonomy issues make full network transparency unachievable. Some have used these issues to argue against any sort of network transparency. The lack of understanding of design choices and of usage patterns described in the previous section makes resolving these conflicting points of view difficult. In this dissertation we address this lack by:

(1) Designing and implementing a prototype of a highly transparent DFS.

(2) Collecting and analyzing data on file and directory references from a large UNIX system.

(3) Using these reference patterns to analyze the effectiveness of our DFS design.

As part of the process of designing a highly transparent DFS we investigate solutions to the various problems faced by a DFS, and examine the interactions of these solutions. This process extends our understanding of the range of possible DFS designs and of the tradeoffs between various designs. The resulting file system design meets the transparency guidelines of our thesis. The prototype implementation validates the design and provides further insights.

The collection of usage data from a UNIX system gives us data on file reference patterns and, for the first time, data on directory references in a hierarchical file system. Analysis of these data

provides detailed information on file system usage.

Finally, the results of the data analysis are used to evaluate our DFS design and to provide guidelines for future designs. This provides further understanding of the applicability of various design alternatives available for DFSs and demonstrates the validity of our original thesis statement.

1.5. The Remainder of the Dissertation

Chapter 2 presents a survey of related previous work. We start by examining existing distributed file system work, particularly as it relates to our goal of full network transparency. We then examine work in areas that are relevant to transparent DFS design. The areas we survey include network naming, mutual consistency algorithms, file placement, file migration, and file and directory reference studies. Based on these surveys we describe areas where significant work remains to be done.

Chapter 3 presents the design of *Roe*, a highly transparent distributed file system. We start by presenting the goals of Roe (in terms of our earlier thesis statement) and describe the environmental assumptions made by the system. We then present an overview of the general architectural principles used in designing Roe, and describe the techniques and algorithms used. Our general approach is to integrate existing solutions wherever possible, extending and adapting where necessary, while maximizing transparency. We include here a discussion of the interaction between the various solutions chosen, and how these interactions drive the choice of algorithms and the architecture of the system. Given this higher level background, we move on to describe in detail the architecture of Roe. We end the chapter with a discussion of the strengths and weaknesses of the approach used by Roe, and evaluate the extent to which Roe meets our goals.

Chapter 4 describes a prototype implementation of Roe on UNIX, RIG, and Xerox Altos. The implementation validates the design presented in Chapter 3 and demonstrates the feasibility of the Roe approach. We start by describing the three environments and the implementation of Roe in each of these environments. We then discuss difficulties that arose during the implementation

(particularly in the area of IPC primitives) and make suggestions for how those difficulties could be addressed in future systems. The remainder of the chapter presents some figures on the performance of Roe and discusses the strengths and weaknesses of the implementation.

In Chapter 5 we turn our consideration to the issue of file system usage patterns. We start by justifying the need for data collection, and then describe a kernel tracing package, implemented under 4.2BSD UNIX, that collects the required data. The rest of the chapter presents the results of an analysis of file reference patterns on a large 4.2BSD UNIX system supporting university research. The analysis includes a breakdown of the data into classes based on file type, file owner, and file user. These cuts allow us to isolate various parts of the file system and user community, and are useful in both understanding the source of references and predicting the behavior of various DFS architectures.

Chapter 6 continues the study of reference patterns, turning to the issue of directory reference patterns. The work presented here confirms earlier conjectures [Ousterhout 85] on the importance of *name resolution overhead in UNIX-like hierarchical file systems*, and provides a basis for analyzing name service in distributed file systems.

Chapter 7 uses the results of our analysis of file and directory reference patterns to investigate some issues in supporting UNIX using distributed file systems such as Roe. We start by examining the availability that one can expect with simple replication techniques, and how this availability may be increased by making use of information on expected usage. Next we consider the implications our measurements have for file placement and migration algorithms. Finally, we then turn our attention to the performance issues our measurements raise.

Chapter 8 briefly summarizes our results and presents suggestions for future work.

1.6. The Significance of This Work

This dissertation advances the field in a number of significant ways

It describes the design and implementation of Roe, a highly transparent distributed file system. Roe supports a substantially higher degree of transparency than earlier distributed file systems, and is able to do this in a heterogeneous environment. Roe provides a coherent framework for uniting techniques in the areas of replication, consistency, file and directory placement, and file and directory migration. It provides the information necessary to allow these techniques to work effectively.

As part of the process of designing Roe we investigate solutions to the various problems faced by a DFS, and examine the interactions of these solutions. This process extends the understanding of the range of DFS designs and of the tradeoffs between various goals and techniques. The resultant design and the analysis of its implementation also extend this understanding.

Another significant contribution of this dissertation is the collection and analysis of file and directory reference patterns. This work is novel in several respects. It is by far the most detailed study of short term UNIX file reference patterns that has been done to date. It is also the only study we have seen that examines the differences between important user and file classes. In addition to examining the overall request behavior, it breaks references down by the type of file, owner of file, and type of user. Knowledge of the substantial differences between these classes will be useful in designing future DFSs and can be used to analyze current DFSs.

In addition, ours is the only study we have seen that collects and analyzes information on directory reference patterns. Our results confirm earlier speculations on the importance of name resolution overhead in UNIX environments and provide information necessary to design algorithms that minimize this overhead, both in single site and distributed file systems.

Finally, the dissertation describes the use of the file and directory reference results to study naming, availability, performance, caching, replication, and initial placement issues in DFSs. These studies provide further understanding of the range of applicability of DFS designs and of the techniques we have used. In addition, these studies demonstrate the validity of the highly network transparent approach used by Roe.

Chapter 2

Previous Work

2.1. Introduction

There has been a considerable amount of work done in distributed file systems and related areas. A substantial body of literature, along with a handful of implementations, exists. While many of these systems provide transparency of one form or another, none has pursued it to the extent that we desire, and there is limited understanding of many of the key issues and tradeoffs. In this chapter we survey previous work and describe how our work fits into the existing framework.

Section 2.2 presents terms and metrics that we will be using to evaluate distributed file systems. Section 2.3 describes a number of existing systems and characterizes them using the metrics presented in section 2.2. The following sections review previous work on issues related to the management of data and resources in a distributed system. The areas surveyed include naming (section 2.4), replication and consistency (section 2.5), reference patterns (section 2.6), and file placement and migration (section 2.7). Section 2.8 briefly surveys other related work. In section 2.9 we summarize what we perceive to be the weaknesses of previous work and describe the general approach we will take in addressing these weaknesses.

2.2. Terms and Metrics

Many measures have been used in the literature to characterize and evaluate distributed file systems. Examples include availability, consistency, transparency, and performance. Definitions of such terms in the literature are often vague and contradictory, or are expressed in a way that reflects most favorably on the system being described. While it is not always possible to avoid this latter form of bias, it is important, when comparing DFSs, to clearly define the basis of evaluation and to state its limits. This is the purpose of the remainder of this section.

We will be using the following terms and metrics in our description of DFSs in section 2.3, in our characterization of Roe in Chapter 3, and in evaluations in Chapter 7:

- Availability
- Consistency
- Performance
- Reconfigurability
- Resource utilization
- Transparency

The intent of this set of measures is to capture and quantify, where appropriate, key distributed file system properties. These measures have all appeared in the literature in one form or another (see, for example [Watson 81], [Saltzer 79], [Cheriton 84], [Lantz 86], [Walker 83a], [Nelson 88]). However, this is the first time that they have been considered together as a means of evaluating DFSs.

2.2.1. Availability

Availability is a measure of the ability of a DFS to provide service despite failures in the underlying hardware. We define availability as the fraction of valid user requests that are successful. Here a valid request is one that could be expected to succeed if all components of a DFS were accessible. For example, enumerating a directory and opening files that are not in use are valid

requests. Attempting to open a file with conflicting access is not a valid request.

Our measure of availability focuses on the behavior of a DFS in the presence of faults. It does not address the issue of operations which fail because of concurrency control and other algorithmic sources. Performing a detailed evaluation of the availability of a DFS requires both knowledge of usage patterns and a model of failures. These issues will be addressed in Chapter 7.

The term reliability is sometimes used in place of availability in the literature.

2.2.2. Consistency

Consistency, for our purposes, is taken to mean that the results of operations are predictable, and that all users see the same result. In particular, write operations either completely succeed or fail without a trace, and later reads show the result of the successful write operations regardless of host failures, network partitions and so on. These two aspects of consistency are often referred to as *atomicity* and *serializability* (section 2.5).

The consistency of a DFS can be characterized by enumerating the conditions under which these properties hold.

2.2.3. Performance

There are two aspects to performance:

(1) *Throughput*: The average rate at which sequences of operations (such as opening a file or reading a block) can be performed.

(2) *Latency*: The mean time that it takes for a single operation to return a result.

In either case, the result is dependent on the operation requested, on the underlying OS and hardware resources, on the configuration of the DFS, and on other operations in progress. Arriving at accurate performance figures and understanding the effects of architectural decisions requires both controlling these factors and having an understanding of the usage patterns presented to a DFS.

2.2.4. Reconfigurability

Reconfigurability is a measure of the ability of a DFS to adapt to changing usage patterns, to failures, and to changes in the configuration of the underlying networks and hosts. Examples of reconfiguration include moving files or requests between hosts to balance resource utilization, incorporating new resources, and rerouting requests to failed components. Factors that influence the reconfigurability of a DFS include the granularity of movement allowed and the information available to aid in decision making.

We will characterize the reconfigurability of DFSs in terms of the granularity of reconfiguration supported, the effectiveness of load balancing algorithms, and the types of failures handled.

2.2.5. Resource Utilization

A distributed file system makes use of disk, network, and CPU resources in servicing requests. Minimizing and balancing demands on these resources allows a DFS to support more users, or can decrease the impact of a DFS on other applications. We use the measure *resource utilization* to capture this aspect of DFS operation. We will be characterizing the resource utilization of DFSs in one of two ways:

(1) Usage as a fraction of the total resources available on a host for each type of resource needed. Resources of interest include disk storage, disk bandwidth, network bandwidth, and CPU cycles.

(2) Usage relative to that of a simple central file server implementation.

The first measure provides useful information on bottlenecks in a design, and on the effect it will have on other users of these resources. The second can be used to estimate the resource cost of various architectural and implementation decisions (for example, the resource overhead of distributed locking or of data replication).

2.2.6. Transparency

A transparent DFS is one that allows objects to be created and accessed by a user without constraints on the name, the location of the user or object, and with no knowledge of the configuration of the underlying system. Transparency has been decomposed as follows [Walker 83a]:

- *Location transparency*: The name of an object does not determine its location.
- *Name transparency*: The meaning of a name is independent of the user's site.
- *Semantic transparency* (also referred to as *semantic consistency*): The meaning of an operation is independent of the site from which the request is issued.

Other aspects of transparency that may be used to characterize DFSs include:

- *Access transparency*: Objects may be accessed in the same way independent of location.
- *Failure transparency*: The masking of failures that occur during use of the DFS.
- *Implementation transparency*: The method or results of accessing an object do not depend on the implementation of that object or of the resources used to support it.

The aspects of transparency that we have outlined here impact on the other measures that we have described. For example, limits on name and location transparency will typically affect the reconfigurability of a DFS, and limits on implementation transparency may in turn limit the techniques used to improve availability. For each of the DFSs presented in section 2.3, we will describe the architectural features that affect various aspects of transparency and also describe, where possible, how this affects other measures.

2.3. Existing Distributed File Systems

This section surveys existing distributed file systems. We do not attempt to provide a complete enumeration of existing DFSs, but rather present a handful that are representative of the approaches that have been used. We emphasize DFSs that attempt to address the transparency, availability, reconfiguration, performance, and user model aspects of our thesis statement.

2.3.1. Helix

A common view of distributed systems is as a collection of services that are available to user applications. This model (often referred to as the client-server model), leads to the notion of file servers that are relatively independent of clients. These services provide file and block storage. Some also provide support for directory structures. Helix [Fridrich 84] is a distributed system that incorporates one example of such a file service.

Helix is organized as a collection of servers that provide capability-based access to objects under their control. Servers manage volumes, which are logically autonomous units of storage. There is typically a one-to-one correspondence between a volume and a physical disk. Volumes contain files and directories. Files contain uninterpreted data that may be read and written by clients. Directories are repositories of capabilities, and allow users to associate names with files and other directories. Capabilities encode information on object location and access control. The intent is that volumes be highly autonomous, so capabilities in a directory refer to objects on the same volume (no cross-linking is allowed).

Helix supports atomic transactions that may include files on multiple volumes, with two-phase commit protocols being used to ensure consistent results (section 2.5). One writer/multiple reader locking is used for concurrency control, with the addition of support for access to snapshot copies of a file [Fridrich 81].

From a client's point of view, the effective availability of Helix is the probability that all Helix servers that the user needs are accessible. There is no support for increasing availability by replicating important resources. This would be difficult in the current architecture, given the requirement that directories refer to objects on the same volume.

This requirement, along with the location-dependent capabilities used by Helix, means that names at both the pathname level and capability level are not location transparent. Objects must remain on the server where they were created. This eliminates any possibility for reconfiguration. On the other hand, this approach ensures that if a server is accessible, all objects on it will also be

accessible.

Two systems that take an approach similar in spirit to the one used by the Helix file service are the Cambridge File Server [Birrell 80, Mitchell 82] and the Xerox Distributed File System (XDFS) [Mitchell 82, Sturgis 80]. The Cambridge File Server is also capability-based, but is optimized for fast, simple transfers and only supports atomicity on a single file basis. XDFS was designed to support database research. It provides byte-level locking and extensive facilities for transaction management.

Svobodova surveyed a number of server-based systems [Svobodova 84] and discussed desirable characteristics of such systems. Her conclusions were that a high level of abstraction, atomicity across multiple files, integration of local and remote storage, and replication are useful file server properties.

2.3.2. NFS

One alternative to a server-based approach is to patch remote directory trees into the local name space. This allows the same access methods to be used independent of where a file is located (access transparency). Sun Microsystem's NFS [Hatch 85, Kleinman 86, Lyon 85] is an example of such a DFS.

NFS allows UNIX file systems on remote machines to be "mounted" in a local UNIX directory, in the same way that local file systems may be mounted. These remote file systems may be mounted at an arbitrary point in the local naming tree, and then are accessed by the user with most of the same operations that would be used to access local files. Each kernel maintains a *mount table* that describes where in the naming tree file systems are mounted. File systems are generally mounted as part of the boot sequence, although they may also be mounted manually.

When a request to access a remote file is made, the local UNIX kernel contacts the remote kernel to read the appropriate blocks. The local kernel maintains a cache of blocks read to minimize server traffic and to increase performance. Updated blocks are also cached locally, and eventually

(within 30 seconds) written back to the remote host. NFS uses stateless protocols to decrease complexity and to simplify crash recovery. No state (beyond caching to improve performance) is maintained on the remote host, and so concurrency control is not supported by NFS. Applications requiring concurrency control for NFS files either implement it themselves or use a separate lock management service.

NFS can, with a careful choice of naming and mounting conventions, provide name transparency for most files (the root file system and some administrative file systems are always local to a host). This, combined with the access transparency described earlier, allows a user to run applications and access files independent of relative user and file locations. Users can take advantage of the file resources of other hosts without needing to know the details of their location.

NFS does not, however, provide complete location transparency. A UNIX file system provides support for a complete subdirectory of the naming tree (excluding other file systems that may be mounted in this subtree). All files in the subtree reside on this file system, and so are bound to the host (and disk) supporting the file system. Binding sections of the naming tree to a file system means that the resources available to users of that tree are limited to those on that file system. Limitations in storage space and processor capacity can crop up in unexpected places. Reconfiguration to correct problems or improve performance is difficult, because of both the forced grouping of files and the relatively static nature of the mount table.

The stateless server approach used by NFS allows it to provide limited failure transparency. If a server fails, clients are simply suspended until the server is available again.

The availability of an NFS-style system is less than that of a host running alone. Users will generally require access to multiple machines to perform their work. The effective availability is the probability that all required machines are accessible.

Consistency (or rather, the lack of it) is also a problem in NFS. NFS delays writeback of updated cached blocks to improve performance. This allows transient cache inconsistencies to arise, and so users on different hosts can see inconsistent data. The stateless nature of NFS protocols precludes

the incorporation of concurrency control mechanisms into NFS.

AT&T's RFS [Hatch 85, Rifkin 86] and COCANET [Rowe 82] also take the approach of mounting remote file systems into a local UNIX naming tree, although both of these DFSs forward requests to remote systems and maintain state in order to retain UNIX semantics.

2.3.3. Sprite

Sprite [Nelson 88, Ousterhout 88] is a network operating system being developed for the SPUR multiprocessor workstation. Sprite provides users with a UNIX-like environment, including a file system that may be shared among machines.

The Sprite file system is organized as a collection of *domains*. Domains are similar to Unix or NFS file systems in that each domain implements a subtree of the name space and all files in a domain reside on a single server machine. Each client kernel maintains a *prefix table* that provides hints on the server supporting a subtree. Lookup proceeds by finding the longest prefix matching a given file name and then forwarding the remainder of the name to the server supporting the prefix. Special *remote links* are used to mark domain mount points. If the location of a server isn't known or if the prefix table entry for a prefix is invalid (because the domain has moved or been replaced), the client kernel broadcasts the prefix to all servers. The server managing the tree replies and the prefix table is updated. This allows domains to be moved dynamically.

A major focus of the Sprite file system is caching. This was prompted by the continued growth in workstation memory size, and by a study showing the performance benefits possible with even a moderate-sized file cache [Ousterhout 85]. The Sprite file system, like NFS, uses main memory block caching at both remote servers and clients. Sprite, however, guarantees that all users accessing a file simultaneously will see a consistent view of the data in the file, even in the presence of caching and multiple concurrent writers. Sprite client kernels contact the server supporting a file on every open and close of that file. This is to both validate locally cached data and to allow the server to detect concurrent sharing of a file. If the server finds that an open for write is requested while other clients are using the file, it disables client caching on the file and forces all operations

to be forwarded to the server. This allows the server to ensure consistency.

The Sprite file system maintains a considerable amount of state for ongoing operations at remote servers. This allows Sprite to provide richer semantics and more efficient operation than NFS, but also results in Sprite not having NFS's failure transparency.

As with NFS, the availability of Sprite is simply the probability that the required servers are up. Prefix tables can allow Sprite to continue operation in some cases even if the server supporting the root domain is unavailable. Sprite guarantees name transparency, since domains are mounted in the same place in the tree by all clients, and there are no local domains corresponding to the local file systems in NFS. The dynamic nature of prefix tables allows domains to be moved from server to server, but the requirement that files in a given domain be moved together limits the reconfiguration possibilities here.

2.3.4. Andrew

Andrew [Morris 86] is a computing environment intended for large scale networks of personal computers. An important part of this environment is the Andrew file system [Howard 88, Satyanarayanan 85]. This file system supports a UNIX-like naming tree composed of file systems local to the client machine (referred to as *Virtue*) and a global name space supported by dedicated Andrew servers (collectively referred to as *Vice*).

The structuring primitive in Andrew is the *volume*. Each volume supports a subtree of the overall global naming tree. All files in a volume reside on the same server. Each Vice server maintains a *Volume Location Database* that dynamically maps volumes to servers. This allows volumes to be moved to balance server utilization and disk consumption. Studies of an earlier version of Andrew showing that utilization varied by as much as 5:1 from server to server demonstrate the importance of having this capability. Volumes in Andrew are typically dedicated to one user and may grow and shrink in size depending on the disk space available on a server and on the needs of the user.

As with NFS and Sprite, Andrew relies heavily on caching. However, Andrew caches on-disk instead of in memory, and caches whole files instead of on a block basis. Caching on disk allows the cache to be larger and to survive crashes. Whole file caching limits the size of files that may be accessed by a client, but UNIX files are generally small and so this is usually not an issue. The decision to cache whole files was motivated by the desire for a system that would scale to large numbers of users. Whole file caching allows interactions with Vice servers to be minimized. When a file residing in Vice is opened, Virtue caches a copy on the local disk and then directs all further operations to the cached copy. When a file is modified, all work is done locally and then a copy of the updated file is stored back on Vice. Clients caching a copy of a file assume that the cache is correct. Vice keeps track of where copies of a file are cached and notifies clients to invalidate their cache if the file is updated. This mechanism doesn't guarantee consistency if a file is used concurrently by multiple clients, but it does support consistency for serial access.

Performance comparisons between NFS and Andrew show that Andrew suffers from higher latency at low loads, but performs much better at high loads, and so can support more users per server. Both of these characteristics are due to the use of *whole-file caching*. Resource utilization of the network, server CPU, and server disk bandwidth are all lower for Andrew.

The Andrew file system supports the replication of read-only volumes. This can be used to provide higher availability for volumes containing system executables and other slowly changing files. There is currently no support for replication of read-write volumes; the availability of these volumes is that of their server.

The "serial access" consistency provided by Andrew is somewhat stronger than that provided by NFS, but weaker than that provided by Sprite. Andrew provides name transparency for files stored in Vice but not for files in file systems local to a client. The volume structure used by Andrew provides for a higher degree of reconfigurability than NFS, despite the limitations on location transparency imposed by the need for all files in a subtree to reside on the same volume.

The Cedar File System (CFS) [Gifford 88] also caches on a whole file basis. The primary use of CFS is to support program development. Concurrent file update is extremely rare in this environment and developers are usually interested in a static snapshot of a set of files. Because of this, it is reasonable for CFS to support only *immutable* files (files that cannot be changed once created). Updating an immutable file results in a new file being created. This neatly sidesteps cache consistency problems.

CFS supports the replication of immutable files on a directory-by-directory basis. This is done using a background daemon that periodically copies files added to a directory to replica servers. The immutability of files makes this a relatively simple process.

2.3.5. LOCUS

The LOCUS distributed operating system [Walker 83a, Walker 83b] emphasizes network transparency, availability, and performance on a local area network. Work has also been done to extend LOCUS to the Internetwork environment [Sheltzer 85].

LOCUS is an extension of BSD UNIX, and so supports a UNIX-like hierarchical file system. The file system is organized around *logical filegroups*. Logical filegroups resemble UNIX file systems, in that each one implements a subtree of the global file space. These subtrees are glued in place using the UNIX mount mechanism. The resulting mount table is stored at each host in the network and is used in pathname resolution and file opens.

LOCUS allows files to be replicated to increase availability. To support replication, LOCUS associates one or more physical containers (UNIX file systems) with each logical filegroup. A file or directory in a logical filegroup may be present in any subset of the containers for its filegroup. Access to files in a logical file group is synchronized by the current synchronization site (CSS) for the file group. Updates are made to a copy of the file selected by the CSS and then propagated to the rest of the copies. Updates to a file are atomic.

LOCUS guarantees consistency of replicated files and directories in the absence of partitions. If a partition occurs that splits the physical container for a file group, CSSs for the filegroup are established in each partition. This ensures the availability of the file group in both partitions, but may allow inconsistencies to arise. LOCUS guarantees that inconsistencies resulting from updates during partitioned operation will eventually be detected (and in some simple cases, resolved [Parker 82]), but provides no means for notifying users that they may be accessing an inconsistent copy.

LOCUS is one of the few existing distributed file systems that attempts to provide support for heterogeneity. LOCUS supports *hidden directories* that stand in for machine dependent files (such as executables). A hidden directory contains a different version of the machine dependent file for each architecture supported. The correct version is selected using a per-process context. It should be noted that the heavy reliance of LOCUS on UNIX would make extensive support for operating system heterogeneity difficult. NFS is another DFS that has attempted to address heterogeneity. This has met with very limited success, both because of NFS's close ties to UNIX and because of the stateless protocols used by NFS.

A major strength of LOCUS is the high availability that results from replication and operation during partitions. This high availability is at the expense of consistency during partitioned operation. The replicated global mount table used by LOCUS generally ensures name transparency, although this can break down in the presence of partitions. The use of the CSS provides implementation transparency. LOCUS provides a somewhat higher degree of location transparency than NFS, since files in a logical filegroup can be located on any of the physical file systems supporting the logical filegroup. This in turn provides a greater degree of reconfigurability. LOCUS does not provide complete location transparency, though (files are bound to the logical filegroup where they are created), and so there are limits to the amount of reconfiguration allowed.

2.3.6. IBIS

The IBIS file system [Tichy 84] takes a completely different approach from that used by LOCUS and most of the other DFSs we have discussed. Rather than patching together a global name space out of a number of subtrees, IBIS supports its own name space. This name space is independent of the physical and logical locations of files and directories. The IBIS global directory supports a UNIX-like hierarchical directory tree. Each node of the directory is a separate IBIS file. IBIS files (and hence directory nodes) are replicated.

Replication is done in IBIS using a primary copy algorithm, with updates going to a copy designated as the primary, and reads going to any copy. When a primary is updated, copies can either be updated or invalidated. There may be some delay between updates being made to a primary and propagation of updates to copies, and so transient inconsistencies can arise. If the primary copy of a file is unavailable, a temporary primary is elected. In the presence of partitions there may be several active primaries, thereby causing inconsistencies. Temporary and actual primaries are merged (if possible) when communication is reestablished.

Directory lookup requests go to the local copy of a directory node if one exists. If a local copy doesn't exist, one is created using the primary copy of the node. Each entry in a local directory contains a pointer to the primary of the file or directory referenced by the entry and to a local copy (if any). Referencing a file also, by default, causes a local copy to be created. This differs from the on-disk cache used by Andrew in that these copies are (optionally) updated as the primary copy changes and may be accessed and updated even if the primary is unavailable.

IBIS provides complete location transparency. Because of this, the IBIS architecture would be able to support a high degree of reconfigurability. IBIS uses location transparency to improve performance by replicating copies of files and directories on nodes where they are used. IBIS also supports a manual operation that allows the location of a primary copy to change (migration), but doesn't provide support for automatic reconfiguration of this sort.

The availability of IBIS depends on the relative proportion of reads to updates, on whether copies of a primary are updated or invalidated on update, and on the depth of directory trees (since each node is another component that must be accessible). IBIS provides name transparency in the absence of partitions. However, in the presence of partitions inconsistencies can arise in both files and directories. The performance of IBIS is difficult to evaluate. The creation of local copies should result in good performance for files with a high ratio of reads to writes. However, performance can be expected to suffer for files with mixed opens for read and write (due to the creation and invalidation of local copies).

2.3.7. MULTIFILE

MULTIFILE [Gait 86] is a distributed file service that runs on an Ethernet-based LAN of engineering workstations. MULTIFILE uses multicast to distribute file open requests, with the first file responding handling the request. The remaining available copies *shadow* this copy (performing operations sent to it on themselves when necessary) and take over if it fails or is too slow.

MULTIFILE provides high availability, implementation transparency, location transparency, and failure transparency. There is no support in MULTIFILE for user-level directories, no consistency guarantees, and no concurrency control. These are all left to applications. Another drawback of MULTIFILE is the relatively high resource utilization implied by multicast and shadowing.

2.4. Naming

Saltzer [Saltzer 79] gives a detailed discussion of centralized directory structures for resolving user-chosen symbolic names into object references. He discusses hierarchical directories, contexts, binding, aliases, and so on. In a distributed environment, additional issues arise. These include generating names in a distributed fashion, providing support for network transparency, fault tolerance, object location, performance, and reconfiguration. Section 2.3 included a discussion of naming techniques used by existing distributed file systems. This section describes other work that is relevant to naming in distributed file systems. We emphasize the transparency and distribution

aspects of this work.

2.4.1. R*

The R* distributed DBMS [Lindsay 81] ensures that object names generated in a distributed fashion don't conflict by embedding the creation site in the name. Names have the form *user@user_site.object_name@creation_site*. The *creation_site* field is also used when looking up objects. Each site maintains catalog information on objects currently at the site. In addition, a pointer to the new site is left behind for objects that were created at the site but have moved. Hence lookup requests sent to the creation site can always return either the entry or the location of the entry.

This approach to name generation and lookup has the advantage of being simple and of requiring little interaction between sites. Drawbacks are that users are required to know the creation site of an object (objectionable both from an administrative and a transparency viewpoint) and that moving an object decreases its availability (both its creation site *and* the new home must be available for a lookup to be guaranteed to succeed).

R* caches catalog information on accessed objects to aid in locating objects and in planning, and to decrease access time. No attempt is made to ensure that cached information remains consistent. Instead, it is treated as a *hint* that will be discarded if incorrect. Catalog entries contain version numbers that are used to verify that the hint is valid.

A local area network DBMS developed at IBM Yorktown [Hailpern 82] uses the R* naming scheme, but treats even the creation site given in the name as a hint. To resolve a name, this system first checks the local catalog, then the local cache (for hints), then the creation site, then "well known" servers, and finally does a broadcast to see if the object exists anywhere in the accessible network.

2.4.2. Caching and Hints

Both of the systems described in the previous section use caching and hints to improve lookup behavior. A study of distributed directory caching in LOCUS [Sheltzer 86] showed that even a relatively small 40 page directory cache gave a surprisingly high hit ratio of 95%. Further, the nature of references and updates in LOCUS was such that maintaining cache consistency introduced little overhead. A more comprehensive study of directory caching in the UNIX environment is presented in Chapter 7.

Terry has studied the effectiveness of caching hints under various conditions [Terry 87], and presents an algorithm for determining effective caching strategies. His approach is to attempt to maintain a given level of cache accuracy (instead of attempting to maintain a high hit ratio), with the desired level of accuracy depending on the relative costs of accessing uncached data and recovering from a bad hint.

2.4.3. Clearinghouse

Clearinghouse, a name service developed at Xerox [Oppen 81], also structures names so that they contain location information. In this case, the location information is logical rather than physical. Names in Clearinghouse have the form *local_name@domain@organization*. These names form a three level naming tree. Names are partitioned by domain and organization. A domain server contains information on all objects in its domain. It also contains a table describing the servers supporting the organization level and organization servers contain information on all domain servers in the organization. This hierarchically structured location information allows a client in contact with any domain server to locate any other domain server (and hence any object referenced by Clearinghouse).

Clearinghouse servers are usually replicated to improve availability. The lack of actual location information in names allows this to be done transparently. Updates to replicated information are propagated using the mail system. This may result in transient inconsistencies. Information from Clearinghouse is regarded as a (usually accurate) hint.

Drawbacks of Clearinghouse for DFS use include its lack of consistency and the fixed structure that it imposes on names.

2.4.4. Cronus

Cronus [Schantz 86] is an object-oriented distributed operating system made up of a collection of services. One of these is a directory service that may be used to catalog any object in the system. The directory service supports a tree structured name space (similar to UNIX). Each object in Cronus is named by a unique identifier (UID) that is assigned to the object when it is created. The directory service provides a mapping from an absolute hierarchical name to a UID. Most objects in Cronus can be migrated, so the UID does not usually contain useful location information. Broadcast is used to find the object with a given UID. Mappings from UID to host are cached to alleviate the considerable overhead of broadcast.

Each node of the directory tree is an object in its own right and can be migrated as desired (a manual operation is provided for this). Directory nodes may also be replicated. The primary goals of the Cronus directory service are high availability and performance. Because of this, there is no concurrency or consistency control for replicated directories, and it is possible for inconsistencies to arise due to partitions or concurrent operations. If this happens, nodes are locked until repaired manually (in practice this rarely occurs).

Cronus attempts to cluster directories to minimize the number of hosts needed to traverse a path. This is done by dividing the directory tree up into a seldom-changed "root portion" and a number of subtrees (this division is referred to as a "dispersal cut"). The root portion is widely replicated and an attempt is made to place directories in a subtree on the same hosts. This helps to increase both performance and availability.

The location transparency supported by the Cronus directory service makes it highly reconfigurable. Directories can be moved as needed to balance load and improve performance, and can be replicated to increase availability. Cronus provides name transparency in the absence of partitions and concurrent updates, but because of consistency limitations makes no guarantees.

The use of a broadcast mechanism in name resolution and object lookup can make these expensive operations. This will be particularly true as the size of networks running Cronus grows.

2.4.5. Structure Free Name Distribution

All of the distributed file systems and name services that we have described so far use the structure of the name space to aid in distributing responsibility for resolving names and locating objects. Terry has proposed a design that is structure independent [Terry 86]. His approach is to divide the name space up into a number of *contexts*, each of which manages names that match some *clustering condition*. Clustering conditions can be any arbitrary function that spans the name space and provide a mapping from name to context. *Context bindings* that map contexts to actual sets of servers are also required. Terry's approach supports a wide range of naming structures and allows responsibility for names to be allocated and reallocated as needed, independent of the structure of the name space. It allows a particularly high degree of reconfigurability.

The Emerald programming environment [Jul 88] uses a combination of forwarding addresses and broadcast to maintain an unstructured name space. When an object is migrated, a timestamped forwarding address describing its new host is left behind. Path compression techniques based on timestamp ordering [Fowler 85] are used to keep the chain of forwarding addresses for objects that move frequently short. If a forwarding address can't be used because a host is inaccessible, broadcast is used to locate the object.

2.5. Consistency and Replication

This section examines the problem of presenting clients with a consistent view of data. There are two aspects to the problem: 1) maintaining a consistent internal view of data; and 2) maintaining mutual consistency between copies of replicated data.

2.5.1. Internal Consistency

Serializable transactions have gained widespread acceptance as a general technique for insuring consistency in the presence of failures and concurrent use of data. A transaction is a series of operations on a set of resources. Transactions are guaranteed to be *atomic*, in that they either execute completely or fail totally, leaving no trace of their actions. Serializability means that transactions appear to execute one after another, with no intermediate state from one transaction seen by another. These two properties ensure that the system always appears to be in a consistent state, regardless of failures and the actions of other software. This greatly reduces the possibility for software faults due to the interaction between components and masks the occurrence of hardware faults by providing automatic recovery.

Serializability is provided in the presence of concurrently executing transactions using concurrency control mechanisms. There are two mechanisms in widespread use: two phase locking and timestamp order. *Two phase locking* mechanisms [Bernstein 82] place a lock on any items referenced by a transaction. Transactions trying to place conflicting locks (for example, a write lock on an item with a read lock) are forced to either abort or wait until the earlier transaction has committed (finished). The transaction is run in two phases, with the first phase acquiring all locks that are needed and the second phase performing any updates required and releasing the locks. This two phase behavior ensures that transactions see either all of the results of other transactions or none of them.

Timestamp order mechanisms [Bernstein 82] place a uniquely ordered timestamp on each transaction. Writes by a transaction are accepted as long as no other transaction with a larger timestamp has accessed the data being written. Otherwise, the write is rejected and the transaction must be aborted and restarted. Similarly, reads are allowed as long as no transaction with a larger timestamp has updated the data. Timestamp ordering performs well when there are few conflicts between transactions, but it can result in excessive aborts and restarts when there are many conflicts or when transactions run for long periods.

Atomicity is typically guaranteed using two phase commit and either logging or shadowing. *Two phase commit* [Lampson 80] is done as follows: When a transaction has finished its reads and updates, a site is picked to coordinate transaction commit. This site contacts all sites that have been updated by the transaction and instructs them to prepare to commit. These sites write results from the transaction to stable storage (typically two locations on secondary storage) so that information on the transaction will survive processor crashes and other faults. This is the first phase. In the second phase, the site coordinating the transaction decides to either commit or abort the transaction, depending on the responses from participating sites, and writes this information to stable storage. It then requests all participating sites to either commit the effects of the transaction, making them visible to other transactions, or to abort them. There are several variations on two phase commit that attempt to increase the probability that a decision can be made even in the presence of a failure of the coordinating site [Dolev 82, Skeen 81].

Two phase commit requires sites to keep intermediate information on the changes made to data, to reliably store the status of a transaction during the commit phase, and to restore the old version of data if the transaction aborts. *Logging* approaches do this by maintaining a log of operations on data, including the decision to commit or abort. If a crash takes place, data values can be reconstructed using the log. An alternative is shadowing. In *shadowing*, all work is done on a copy of the data. This copy replaces the original data at commit time.

Nested transactions [Moss 81] are an extension of the transaction mechanism described above. With nested transactions, an application can be composed of a hierarchy of multiple transactions, with these transactions sharing data (using lock inheritance rules that ensure serializability) and committing atomically. There are provisions in nested transaction models for the overall transaction to commit even if some subtransactions fail.

2.5.2. Mutual Consistency of Replicated Data

Replication (maintaining multiple copies of an object) can be used to increase availability and, under some conditions, performance. This raises the problem of insuring that copies contain the

same data (that they are mutually consistent). Methods for doing this generally fall into one of three categories: unanimous update, primary copy, or voting. *Unanimous update* algorithms allow reads from any copy, and force updates to be propagated to all copies. If a copy is down, updates are saved and applied when it becomes available. This approach, used by SDD-1 [Hammer 80], allows efficient access to replicated data when there is a high proportion of reads, but doesn't preserve consistency in the presence of partitions.

Primary copy algorithms [Garcia-Molina 82, Stonebraker 79] elect a single copy to receive both reads and writes, with the remainder of the copies being updated by the primary and acting as backups. If the primary fails, the remaining copies elect a new primary. Primary copy, by itself, doesn't preserve consistency in the presence of partitions. If a further requirement is added that the majority of copies are present in a partition for operation to continue, consistency is preserved.

Voting algorithms [Gifford 79b] assign some number of votes to each copy and require that read and write operations collect some number of votes (the read and write quorums, respectively) before operations can proceed. Setting the sum of the read and write quorums to be greater than the total number of votes ensures that consistent data is always seen. Other algorithms can have performance advantages over voting in many configurations, but voting operates correctly even in the presence of partitions. There are several variations on voting that attempt to increase availability for common sequences of failures [Davcev 85, Jajodia 87] or that make use of type specific information [Herlihy 84].

2.5.3. Relaxing Consistency Requirements

There are a number of other mutual consistency algorithms that place restrictions on the types of failures that may be seen, or that relax consistency requirements. *Available copies* [Bernstein 84] and *regeneration* [Pu 86] are both unanimous update algorithms that assume the absence of network partitions. In available copies, copies that are found to be inaccessible are marked invalid and updated when the site holding them rejoins the network. Regeneration goes even further and just discards inaccessible copies, creating replacements on accessible nodes. The *version vector*

scheme used in LOCUS [Walker 83b] allows inconsistencies to occur during partitions, but uses information contained in version vectors associated with each copy to detect inconsistencies and, if possible, resolve them.

2.6. File System Reference Patterns

Knowledge of file and directory reference patterns is useful in both the design and operation of a distributed file system. During the design of a DFS, reference information helps in understanding the effect of architectural design decisions. In an operational file system, reference patterns can be used in reconfiguring to improve performance, availability, and help meet other goals.

Studies of file system reference patterns can be grouped into three areas: long term reference studies, short term file reference studies, and directory reference studies. Previous work in each of these areas is surveyed below.

2.6.1. Long-Term Reference Studies

Early studies of file reference patterns concentrated on long term (days or weeks) reference patterns that could be used in designing archival migration policies.

Stritter collected information on reference patterns in an IBM mainframe environment [Stritter 77]. Over a period of a year he recorded, for each file on the systems studied, whether or not it had been referenced in a given day. He found that there were no obvious long term trends in access rate, no strong correlation in interaccess intervals, and that the interaccess intervals could be fit by an exponential distribution. Smith analyzed these data in greater detail [Smith 81a] and found that time since last reference, file size, file type, and file age were useful predictors of next access. Lawrie et al. have collected similar usage data from a Cyber system [Lawrie 82].

Satyanarayanan took a static snapshot of the file system on a DEC-10 and used it to study recent file access history [Satyanarayanan 81]. He collected information on file age vs. size and type. He found that most files were small (5 blocks or less) and were generally not very old (1 month or

less since the last change). He found significant differences based on the type of file. In particular, source files tended to be smaller and used longer than output files, program source was used longer than document source, and short files tended to be used longer.

2.6.2 Short-Term File Reference Studies

Recent work has concentrated on short term (on the order of seconds) file reference patterns. This has been motivated by the proliferation of local area networks. The relatively small time delay and high bandwidth of LANs makes migration on a much smaller time scale feasible. Some of this work has focussed on block level reference patterns for use in block caching [Smith 85] or improving file organization [Hu 86]. More interesting from the point of view of a distributed file system designer are studies of logical (operation level) reference patterns.

Porcar studied what was primarily batch activity on IBM mainframe systems [Porcar 82]. The data he collected included when the file was opened, the user, the fraction of the file accessed, and the size of the file. He found that few opens resulted in the entire file being read and that interopen times were short, with more than 80% being less than an hour. He was able to group files by type (temporary, permanent, shared, and system) and found, as with earlier studies, substantial differences between the classes.

Satyanarayanan measured short-term reference patterns on an interactive DEC-10 system [Satyanarayanan 83]. He collected histograms of interarrival times and used them to construct a synthetic driver for a file system simulation. This information was for the system as a whole, and contained no information on individual files. He also collected information on lifetime and on the fraction of open requests for system, temporary, and user files. He found significant differences in read/write rates and lifetimes between the three classes. For example, 2/3 of opens were for read, but only 4% of opens for writes went to system files; 2/3 of opens for write went to temporary files; system files lasted almost forever; and temporary files rarely lived more than a day. These results indicate the importance of taking into account the purpose of files in making placement and migration decisions.

Ousterhout et al. collected and analyzed data on 4.2BSD UNIX file reference patterns [Ousterhout 85]. They collected a trace of open, close, seek, unlink, truncate, and execve operations and then used them to derive information on read/write characteristics, file sizes, and data lifetimes for files in this environment. The authors found that the majority of files were small (a few KBytes) and accessed sequentially. Most were open only a very short time (less than 0.5 seconds) and had a short lifetime (80% lived less than 200 seconds). Their study did not include paging activity, inode access, and directory lookups. They noted, however, that "directory lookups appear to account for a substantial fraction of all file system activity," and that the impact of directory lookup overhead could be expected to increase as block and cache size continue to increase.

The data collection package used by Ousterhout et al. has been extended by Zhou et al. [Zhou 85]. Zhou's package collects detailed trace information on opens, closes, reads, writes, file renames, deletes, process forks, executes and exits. Directory operations, symbolic links, paging, and inode activity are not traced.

2.6.3. Directory Reference Studies

None of these previous studies collected information on directory access patterns. This information is not needed in systems that are concerned primarily with migration to manage disk storage, since files are typically much larger than the directories that reference them. However, a DFS may also migrate and replicate directories to improve performance and availability. In a DFS with non-trivial directory structures, the overhead of directory access is an important performance consideration. Evaluating directory design decisions in the absence of data on reference patterns is difficult.

The only directory reference information that we have has been collected incidentally in other studies. Leffler found that 40% of BSD UNIX system call overhead was due to name resolution [Leffler 84]. Sheltzer et al. found that half of all network traffic in LOCUS was in support of name resolution [Sheltzer 86]. These results demonstrate the importance of directory reference

overhead. Mogul found that an average of 13 entries were searched per lookup in BSD UNIX systems [Mogul 86a], suggesting that directories are small, thus easily moved.

Shelzer et al. also measured directory read/write ratios and investigated directory caching as part of an effort to extend LOCUS to the Internet. They found that only 2.5% of directory references were writes, and that directory references tended to be highly localized.

Chapters 5 and 6 examine the issue of short-term file and directory reference patterns in more detail, and present the results of studies we have made.

2.7. File Assignment and Migration

A significant amount of work has been done in the database field on the problem of placing replicated data in a network to minimize various costs associated with accessing and maintaining data. This problem is usually referred to in the literature as the file assignment problem (FAP) or as the initial placement problem. If data may be moved after being assigned to a location, the related problem of deciding when and where to move, or migrate, data arises. We consider each of these problems in turn below.

2.7.1. File Assignment

The file assignment problem is usually formulated as the problem of selecting file locations to either minimize cost or maximize performance. Storage, communication, and update costs are examples of costs that algorithms attempt to minimize. Algorithms that address performance issues typically attempt to either maximize throughput or minimize response time. In the database environment both of these approaches generally attempt to provide an optimal solution over the lifetime of the assignment. The benefits of incremental improvements here make finding optimal solutions reasonable.

Dowdy and Foster [Dowdy 82] provides a comprehensive survey of solutions to FAP. They note that even with significant simplifications, FAP is NP-complete (by transformation from Vertex

Cover [Garey 79]. Typical simplifications include assuming infinite processor, storage, and network capabilities to allow sets of files to be considered independently, and assuming Poisson arrival rates to simplify analysis.

An optimal solution is not always needed. If files can be migrated later on to adjust for changing usage patterns, if files have relatively short lifetimes, or if quick placement is required (an important characteristic in a distributed file system), an approximation heuristic will be a better choice. A number of approximation heuristics have been developed. Jenny uses methods from graph theory to find solutions with small communication costs [Jenny 82]. Wah creates and searches a graph analogous to a game tree [Wah 80]. Murthy et al. describe a cost-minimizing algorithm based on maximizing the incremental benefit of each new placement [Murthy 83]. Bannister and Trivedi suggest making each new file assignment to the most lightly loaded servers to minimize access time [Bannister 82]. Barbara and Garcia-Molina present heuristics for the assignment of votes to maximize availability [Barbara 86].

2.7.2. Migration Algorithms

File migration can be used to improve the initial assignment of files. It would typically be invoked whenever there is a change in the parameters upon which the initial placement decision was based. For example, usage patterns of a file change over time and also provide more accurate information than that used for initial placement. File creation and deletion change the available space on a device. Users change locations.

Early work on the file migration problem was concerned with developing algorithms for migrating to archival storage files that are no longer being used. The long term file reference studies that we described in the previous section [Lawrie 82, Stritter 77] were used as a basis for this work. Studies on the effectiveness of long term file migration algorithms [Lawrie 82, Smith 81b, Stritter 77] indicate that, without detailed information on file access patterns in the system, using a function of file size and time since last reference (STWS) to decide when to migrate a file produces good results.

Porcar used the data that he collected to study the effectiveness of short-term migration algorithms for files shared by multiple users [Porcar 82]. His goal was to minimize network traffic and delay. He found that multiple copy algorithms that attempted to control the number of copies by minimizing a cost function incorporating the cost to store a file and a dynamic estimate of the cost to update the files produced the best results.

Sheng investigated file migration as a way to decrease file storage, communication, and I/O access costs [Sheng 86]. She assumed infinite network and site capacity to allow files to be considered independently. She developed two types of policies: unrealizable optimal policies based on Markov decision models (taking into account both past and future requests to the file) and polynomial time heuristics that estimated future usage based on an exponentially decaying history of past references. Simulation studies using Poisson-based arrival distributions showed a 5% to 10% improvement over the optimal initial file assignment. One would expect to see greater improvements as the locality of reference to files increased.

The Emerald programming environment [Jul 88] stands alone in its use of automatic object migration. Emerald provides migration on first reference. Studies of simulated mail traffic on Emerald showed a decrease in network traffic of 34% and a decrease in execution time of 22% over the non-migrating system. These substantial improvements demonstrate the benefits that can be realized with even very simple migration schemes.

2.8. Other Relevant Work

Other work relevant to distributed file systems has been done in the areas of accommodating heterogeneity, interprocess communication, scaling, and distributed system instrumentation and modelling.

The file system for the Xerox Alto [Thacker 79] associated a property page with each file that described, among other things, the type of the file. This information was used by the Alto FTP program to perform conversions when transferring files to dissimilar hosts. More recently,

properties have resurfaced as general means for associating information with files [Mogul 86b].

A capability-based network IPC developed at CMU and Rochester [Moore 82] provided support for heterogeneity using strongly typed messages, with type conversion done by the IPC at machine boundaries. An alternative approach, embodied in Sun's XDR [Hatch 85] and Cronus's Canonical data types [Dean 87] is to pass data in a machine independent form, with conversion done at the application level.

Other approaches that have been used in accommodating heterogeneity include using network services to loosely integrate heterogeneous systems [Black 85], hiding heterogeneity using common interfaces, and legislating the problem out of existence. Notkin et al. briefly describe each of these approaches [Notkin 87].

Recent work on high performance IPC mechanisms [Birrell 84, Cheriton 83, LeBlanc 84] has resulted in implementations with machine to machine message passing times of a few milliseconds or less on current hardware. These times are much le s than the cost of a typical disk access (30 milliseconds with current technology) and so it becomes quite reasonable to access data remotely.

The Andrew project that we described in section 2.3 has, as a primary goal, the ability to scale to very large environments. The Xerox Grapevine name, mai , and authentication service [Birrell 82] had a similar goal Experience with Grapevine [Schroeder 84] showed that, while Grapevine was generally successful in meeting this goal, knowledge of the structure and state of the network and of usage patterns would have helped.

Three general approaches have been used in investigating the properties of distributed algorithms and systems: analytic techniques, simulation, and implementation in testbed and production environments.

Analytic techniques are appropriate in cases where the mechanism being studied is fairly simple, it can be examined in isolation, and simplifying assumptions have little impact on the final result. One area where analytic techniques have been widely used is in estimating the availability of

various replication algorithms. Examples include the k-out-of-N model, which has been used to calculate the availability of the regeneration and available copies replication algorithms described in section 2.3 [Pu 86] and the use of state and assignment enumeration to evaluate weighted voting [Barbara 86, Smith 84].

Analytic approaches are intractable for complex mechanisms and in cases where the interactions between components in a system are important. In this case, simulation models are often used. Two approaches are commonly used: queuing theoretical models and detailed simulation. Queuing models have found frequent use in modelling distributed system performance [Goldberg 83, Lazowska 86] and availability [Dugan 86]. Detailed simulation models have been used to study performance [Satyanarayanan 83], availability [Noe 86], concurrency control [Bloch 87], and many other system properties. Both approaches described here are most useful if they can be parameterized using data collected from an existing distributed system.

There is a very real danger when using the methods described above of oversimplifying the system being studied. This can be avoided by instrumenting and measuring an actual implementation in a testbed or production environment [Cheriton 83, Howard 88, Kohler 83].

We will be using a combination of analytic techniques, measurement of production systems, and prototype implementation in our studies. Analytic techniques allow us to easily compare various algorithms. Measuring production systems will give us information on file system usage that will aid in evaluating designs. A prototype implementation will provide validation of analytic results and give us a context for interpreting the effects of various usage patterns and architectural decisions.

2.9. Discussion

This chapter has presented an extensive survey of work in distributed file systems, and of data and resource management issues applicable to distributed file systems. This survey shows that, while there are distributed file systems that provide various degrees of transparency, none is able to do

so in a way that allows users to ignore the underlying network. IBIS comes closest in this regard, but lacks facilities for ensuring consistency, for accommodating heterogeneity, and for effectively managing data and network resources. Other systems, such as Sprite, are able to ensure consistency, but bind files together in ways that can make the underlying allocation of files painfully obvious and that limit the possibilities for reconfiguration. While there are powerful techniques available for supporting transparent naming, ensuring consistency, increasing availability, and reconfiguring to increase performance and decrease costs, there is no general DFS framework available for making use of these techniques. There is no real understanding of the interactions of these techniques, of their range of applicability, or to what extent they can be used to realize a highly transparent distributed file system.

We investigate these issues and address the shortcomings of existing DFSs in the context of the design and implementation of Roe, a highly transparent distributed file system. Roe provides a general framework for supporting the data management techniques that we have described. The design of Roe is described in Chapter 3, and the prototype implementation is described in Chapter 4.

We have also seen that there are limited data available on short term file reference patterns. Studies of long term reference patterns and of short term reference patterns in batch environments show dramatic differences between classes of files, but all existing short term data for interactive environments either fail to distinguish between classes or treat files in classes anonymously. There is, in any case, little data available to aid in understanding usage patterns in interactive environments. There are no data available at all on directory reference patterns, despite indications of the importance of name resolution in file system overhead. This makes it impossible to understand the tradeoffs between various designs.

We address this lack by collecting and analyzing data on file and directory references from a large UNIX system. These studies are described in Chapters 5 and 6. We use the results of these studies (in Chapter 7) to analyze the effectiveness of our DFS design and to explore general issues that arise in supporting UNIX using a DFS.

Chapter 3

The Architecture of Roe,

A Transparent Distributed File System

3.1. Introduction

In this chapter we describe the design for Roe, a highly transparent distributed file system (DFS) for a heterogeneous local area network. Roe presents to users the appearance of a single, globally accessible file system. It uses a replicated global directory, automatic file placement and migration, file replication, and atomic transactions to provide available, consistent, and distributed files.

Unlike the DFSs we described earlier, the approach used by Roe allows it to provide full network transparency, guarantee file consistency, and reconfigure itself to adapt to changing demands and resources. These characteristics allow the user to ignore the presence of the underlying network when creating and using files. This greatly simplifies the use of a network.

The environmental assumptions made in the design of Roe are described in section 3.2. Section 3.3 outlines the goals and guidelines that helped determine the design of Roe. In section 3.4 we sketch out the general approach used by Roe and examine more carefully issues that affected the design. Section 3.5 presents an architecture to implement this approach. Section 3.6 describes the

advantages and weaknesses of the Roe approach and section 3.7 briefly summarizes our results.

3.2. Environmental Assumptions

Roe is designed to run on a heterogeneous local area network. The design was motivated, in part, by the hardware and software available on the University of Rochester Computer Science Department's network. In this section we describe the characteristics of this and similar environments that have had an influence on the design of Roe.

Heterogeneity, at both the hardware and software level, is an unavoidable characteristic of our environment. The rapid advance in the state of the art in computer hardware, combined with incremental network growth and the differing needs of various classes of users on a network, ensures this. Dealing with heterogeneity by enforcing a common hardware or software base is not an effective solution in such an environment. We assume that the network hosts are made up of a mixed collection of servers, workstations, and general time-sharing machines. These hosts will typically *not* run a common operating system. This means that mechanisms that are intimately tied to those provided by any given operating system will be inappropriate.

We further assume that these hosts are connected by a high bandwidth, low delay network. An example would be an Ethernet [Metcalfe 76] or several Ethernets connected by high speed gateways. While individual hosts or gateways may fail, we assume that the network itself will generally be available. We allow for the possibility of a network partition, but not for a total failure of the communication medium or for hosts that become detached from the network. Our assumptions of high bandwidth and low delay, along with a highly available network, make it both reasonable and appropriate to consider remote access to resources.

The underlying interprocess communication (IPC) mechanisms are assumed to support asynchronous typed message passing, long term connections, notification of connection failure due to host failure or network partition, and some means of naming and locating remote processes. These requirements are all met by various existing IPC mechanisms [Moore 82, Sansom 86].

We assume that there is low write contention for any given file. Files in our environment typically either belong to a single user or are normally read-only files. While this assumption is appropriate for our environment (see Chapter 5 for details), it would not, of course, apply in a shared database environment.

Finally, we assume that all hosts that will be supporting and using Roe are under a single administrative control, and that autonomy (control over local resources) is not an issue. These assumptions are generally appropriate for hosts that are connected to a local area network, and allow us to freely share the resources on the network.

3.3. Goals of the Roe Distributed File System

Roe is intended as a demonstration of the validity of our thesis statement:

> Full network transparency in distributed file systems offers significant benefits. These benefits include increased availability, more effective use of resources, the ability to adapt to changing demands, transparent reconfiguration to adjust to changes in resources, a greatly simplified file system model from the user/application point of view and, with careful design, enhanced performance over other DFS approaches.

Using this thesis statement and the environmental assumptions presented in the previous section, we can construct a more concrete set of goals for Roe. These goals are as follows:

- **Network transparency**: Our thesis statement argues that Roe should support complete network transparency. This includes, as described in section 2.2.6, access transparency, location transparency, name transparency, semantic transparency, failure transparency (for at least some classes of failures), and implementation transparency. The intent is that the user need not be aware of any network related characteristics of accessed files.
- **Simple user model**: One potential advantage of network transparency is that it can free users from the need to understand the details of the underlying network. This allows

them to develop a simple model of their environment. We ask that Roe support this style of use. Roe should allow users to create and access files with no constraints on the name, the location of the user or file, and with no knowledge of the underlying hosts and networks.

- **Consistency**: A related goal is that of consistency. Previous experience in building distributed systems has shown that a lack of consistency makes dealing with distributed systems awkward and error prone for both users and applications. Because of this, we ask that the results of operations be predictable and that all users see the same results. This further simplifies the user's view of the system, eases the implementation of applications that use Roe, and is necessary to support name transparency.
- **Enhanced availability**: The decentralized nature of a local area network, with its localized failure points, increases the probability that at least some resources will be available in the presence of failures. Roe should be structured in a way that takes advantage of this characteristic to enhance the availability of the data it manages.
- **Reconfigurability**: The network transparency supported by Roe will allow files and directories to be moved and distributed without user knowledge or intervention. This might be done, for example, to incorporate new resources, avoid failures, improve performance, or to dynamically balance load in response to changing demands. We ask that Roe support the ability to reconfigure files and directories in this manner and that it collect and maintain the information necessary to do this effectively.
- **Performance**: Because of the widespread use of distributed file systems in interactive environments, it is important that the delay perceived by users be small.
- **Heterogeneity**: Roe should take into account the heterogeneous nature of the underlying systems. This includes differences in the hardware and software base, performance, capacity, and availability of each host.
- **Scalability**: Local area networks range in size from networks connecting a few hosts to those connecting hundreds or thousands of hosts. It is particularly important in the larger and more complicated networks that the user receive help in using network

resources. We ask that Roe scale to networks of this size without significant degradation.

In addition, the following guidelines have helped determine the design of Roe:

- **Use of existing host operating systems**: The heterogeneous environment supporting Roe makes it impractical to undertake a significant implementation for each new operating system and hardware base encountered. A decision was made early on to make use of the existing file systems of each host.
- **Testbed for experimentation**: We expect that Roe will be used as a tool for exploring the interactions between various file management algorithms. This has led us to use an approach that provides a framework for incorporating a variety of algorithms, as opposed to one that is tailored to a particular approach. In addition, Roe attempts to collect information general enough to be used by a wide range of algorithms.
- **Decentralized control**: Our desire for a design that scales to large local area networks has led us to investigate and incorporate algorithms that distribute control of data to locations where the data reside and are accessed.
- **Operation in the presence of partial knowledge**: It becomes exorbitantly expensive in a large network to maintain complete knowledge of the state of the network and its resources. For this reason we have emphasized algorithms that do not require complete knowledge of the network state.

It is important to recognize that the goals we have specified interact and, in some cases, conflict. For example, algorithms that ensure consistency do so at the expense of availability and, in many cases, performance. One measure of the success of Roe will be the degree to which it can meet these conflicting goals. We will return to this issue at the end of this chapter, and again in Chapter 7.

3.4. The Roe Approach

3.4.1. Overview

This section is an overview of the approach used by Roe to meet the goals we have outlined. The remainder of the chapter provides more detail on the major aspects of the Roe design. The emphasis in this chapter is on the general techniques Roe uses to meet its goals, and on how these techniques interact. Chapter 4 will describe an implementation of Roe based on the techniques.

Single site file systems generally present to users an abstraction of consistent, shared files. This allows information to be easily shared between users and their applications. Roe preserves this model across the network. It supports a single, globally accessible file system that provides highly available, consistent, distributed, sharable files in a heterogeneous environment. The use of this model allows users to ignore the presence of the network that supports Roe, with network details being handled by Roe. This greatly simplifies the use of a network.

Roe uses file replication, atomic transactions, a replicated global directory, a detailed model of the network, automatic placement, and migration to provide full network transparency. It runs on top of existing operating systems and uses the resources of the existing heterogeneous hosts for storage.

File replication is used to enhance the availability of Roe files and directories. Weighted voting and atomic transactions are used to ensure that users see a consistent view of files and directories. Weighted voting was chosen for its simplicity, its support for decentralized control, and its ability to operate in the presence of incomplete knowledge. This choice is central to the design of Roe. Our motivation for choosing weighted voting over other alternatives is discussed more fully in section 3.4.2.

Roe maintains a replicated global directory that is used to name Roe files and directories. This name space is separate from the name space of the hosts that support Roe. This separation allows Roe to transparently replicate, distribute, and reconfigure files and directories. The global

directory is replicated using a modified weighted voting algorithm. We describe it in more detail in section 3.4.3.

A network model encodes information used in placing and migrating files and directories. The model includes information on the hosts on the network, their up/down state, available space, and so on. Section 3.4.4 provides information on this model.

Roe uses automatic file placement (section 3.4.5) to free users from the need to specify locations for newly created files. Files are placed based on available space, congestion, topographic and other considerations. File migration (section 3.4.6) is used to change the locations of these copies. This would typically be done to improve performance by moving data closer to users or to balance load or adjust to changes in underlying network resources. The Roe design also supports automatic directory placement and migration.

A network is typically made up of a number of dissimilar machines, and Roe recognizes this by providing extensive support for heterogeneity (section 3.4.7). This includes use of existing host operating systems (to minimize the cost of incorporating new host types), local servers with uniform interfaces, type conversion between machine boundaries, and support for machine-dependent files in the cases (such as executable files) where automatic conversion isn't practical.

3.4.2. File Replication and Consistency

3.4.2.1. Motivation

Roe replicates files and directories. This allows it to provide increased file and directory availability without special hardware support. Unlike earlier file systems we have seen with support for replication (for example, LOCUS [Walker 83b] and IBIS [Tichy 84]), Roe guarantees a consistent view of these replicated resources. Enforcing consistency in this context allows Roe to provide network transparency. In particular, consistency ensures that each user will see the same results, irrespective of location, and that earlier results will not reappear because of host or network failures. This, in turn, eases use of the system by both users and applications.

Consistency in this context has two aspects: keeping a copy internally consistent, and maintaining mutual consistency between replicated copies. The following two sections consider these aspects of consistency, and describes how Roe maintains file consistency. Section 3.4.3.2 examines the issue of maintaining consistency in replicated directories.

3.4.2.2. Internal Consistency

We can ensure that users see a consistent view of a file copy at any given time by serializing access to the copy. It is also necessary to ensure that any changes made are applied atomically, and that these changes be coordinated with the changes made to other copies. This leads us to use *serializable transactions*, as described in section 2.5.1.

Roe is intended for an environment where update sharing of a given file is infrequent, and where conflicts seldom arise. Hence it is appropriate to treat the entire file as the object to be updated, and to serialize access to the file as a whole. Of the two serialization methods discussed earlier (locking and timestamps), locking (at the site of the file) appears most appropriate, given the large size of the object to be accessed, the potential for long periods of access, and the low level of conflicts. The actual locking scheme used by a site is of little interest to Roe, as long as it serializes access (e.g., R/W locks are enough, but a site could use R/I-W/C locks [Gifford 82] to allow more concurrency). Locking requests are rejected if they cannot be immediately satisfied.

A two-phase commit protocol [Lampson 80] is used to ensure that writes are applied atomically and that updates are coordinated. The choice of logging vs. shadowing to implement intermediate storage and support recovery is largely irrelevant to Roe. The decision on which to use on any particular host is best determined by the operating system support available.

3.4.2.3. Mutual Consistency

We desire a mutual consistency algorithm that improves availability, has good performance, imposes minimal requirements on the structure of Roe, and behaves correctly under partitioning and other failures. In more detail, the factors we consider are:

- **Availability**: Replicating files should increase their availability.
- **Performance**: We wish to minimize the delay perceived by users and also the total amount of activity.
- **Flexibility as a Testbed**: We desire an algorithm that does not impose undue restrictions on file placement, migration, reconfiguration, and other algorithms and techniques used by Roe.
- **Behavior under partitioning**: The algorithm should guarantee consistency under partitioning and other partial failure modes.

There are 3 general classes of methods for insuring mutual consistency between copies: unanimous update, primary copy, and voting (see section 2.5). In addition to these three classes, there are algorithms that provide consistency under certain circumstances, or that relax the consistency requirement. Examples of the former include regeneration [Pu 86] and available copies [Bernstein 84], both of which provide consistency in the absence of network partitions or disconnections. The version vector scheme used in LOCUS [Walker 83b] is an example of an algorithm that relaxes consistency requirements, attempting to compensate for inconsistencies at a later time. Since we are requiring consistency in all situations, we will not be considering algorithms in either of these classes.

Unanimous update algorithms write to all copies of a replicated file when making an update. These algorithms actually decrease the availability of a file in our environment and so will not be considered further.

Primary copy algorithms [Stonebraker 79] designate one copy to be the primary at each point in time. All reads and writes go to the site of this copy, and it is responsible for notifying other copies of changes. As long as the primary site is up, reads and writes may continue (one usually also requires that a majority of the sites be accessible to ensure consistency during partitions). If the primary site fails, an election [Garcia-Molina 82] is held to decide on a new primary copy, with copies communicating among themselves to select a new primary. For example, Stonebraker determines the primary copy based on the status of the network (which sites are up) and a fixed

linear ordering of the copies of an object. To determine the primary copy, each site accumulates a list of up sites and then checks to make sure that all agree on the up-list and hence come to the same conclusion about the identity of the primary. This method requires that all live copies be current. This can be handled by postulating an underlying message transmission system that can buffer update messages for later transmission to a down site, or by transmitting an update log to a site when it rejoins the network.

Primary copy, with the majority requirement described above, provides the strong consistency desired by Roe, even in the presence of partitions. A file replicated using primary copy will be accessible as long as a majority of sites are up, and so availability will be enhanced under normal conditions.

An attractive feature of primary copy is its performance. If the location of the primary copy is known and it is up to date, opening a file requires contacting just the primary copy of that file. Under these assumptions, 2 messages are required to open the file, and the delay is $2d$ (where d is the one way message delay and we ignore the delay for any disk activity that might be required). However, if the primary copy is not known, or if it is inaccessible, an election, which is an expensive operation in terms of message activity, is required to open the file.

All solutions that can be classified as primary copy share two important characteristics: the need for agreement on a single authority governing the object (e.g. the identity of the primary copy) and a method for insuring the currency of candidates that may take over the primary role. In our environment, with the algorithm we have described, this reduces to the requirements that each copy of an object reliably knows the locations of all other copies, and that all accessible copies be kept up to date. The requirement that each copy know the locations of other copies makes migration difficult, since all copies must be be informed when one moves. Each list of copies must agree in order for the determination of the primary copy to work correctly. Creating new copies (e.g., caching a temporary copy to increase performance) also requires that all other copies be informed. The currency requirement also interacts with migration. A reasonable restriction in this situation would be that a copy could migrate only when the file was not opened for writing.

Voting algorithms, on the other hand, associate vote information with each copy of a file and allow or disallow actions based on collecting this voting information from each copy. We will consider here a generalized form of voting known as weighted voting [Gifford 79b] applied to file opens. Weighted voting associates with each copy of a file a timestamp and some number of votes. A read quorum, r, and a write quorum, w, are defined for the overall replicated file. When a file is opened, the timestamp and votes are collected from the copies. At least r votes (the read quorum) must be collected to read a file and MAX[r,w] to write it. Reads can be from any current copy and writes go to current copies which hold a total of at least w votes (the write quorum). Making $r + w$ greater than the total number of votes in all copies of the file ensures that at least one current copy will be in any quorum. The timestamp of each participating copy is incremented when the copy is updated.

Weighted voting provides the strong consistency desired for Roe. The vote collection procedure ensures that consistency is preserved even in the presence of network partitions.

Weighted voting allows the number of votes held by each copy to be adjusted based on host availability. This can be used to increase the overall availability over the unweighted case [Garcia-Molina 84]. In addition, read and write quorums can be adjusted to favor commonly executed operations (Chapter 7), and variations of weighted voting exist that can allow even update operations with fewer than a majority of copies [Jajodia 87]. These factors will allow weighted voting to provide higher availability than primary copy in many cases.

A drawback of weighted voting is the relative complexity of file opens. Assuming a multicast open protocol, with open requests being sent out to each of the n participating copies in parallel, an open now takes $2n$ messages. However, the time delay remains $2d$ (plus any queuing delays resulting from the multicast message traffic). Although the initial activity is fairly high, the delay perceived by the user is comparable to the primary copy algorithms. For our purposes, n will usually be a small number (one to three).

Voting solutions do not require that a copy be brought up to date when a site recovers. The existence of obsolete copies is acceptable as long as at least a write quorum of current copies exist. The version number associated with each copy allows obsolete data to be easily detected and updated. It also isn't necessary for each copy to know of the locations of other copies, or for any central authority to have an accurate view of where copies are located at any given time. As long as a quorum of copies is reachable, operations can proceed normally. These two characteristics will simplify algorithms for caching and migrating copies, since they relax currency requirements for data affected by these operations.

As we have seen above, primary copy can be characterized by the need for agreement on a single authority, which in practice leads to a requirement that candidates for primary copy be current and that each copy has a current and correct view of the locations of other copies. Voting, on the other hand, is able to tolerate out of date copies, and can operate with partial or out of date knowledge of copy locations. Our desire for decentralized control, operation in the presence of partial knowledge, and our use of migration for reconfiguration leads us to select weighted voting as the basis for mutual consistency control in Roe, despite the greater cost it imposes for some operations. Weighted voting provides the desired consistency and availability properties without requiring expensive state maintenance in the presence of caching, migration, network partitions and host crashes.

3.4.3. The Global Directory

Unlike most other distributed file systems we have seen, Roe implements a global directory that is independent of the directories maintained by the hosts that support Roe. Information in this directory is used to translate operations on a Roe file into operations on individual copies of the file.

Retaining control over the directory allows Roe to transparently place, migrate and replicate both files and the directory itself. It also allows Roe to easily adapt to changes in the underlying resources (for example, the addition of a new server). This can be contrasted with systems, such as LOCUS and NFS, that patch together existing naming subtrees. This patching makes it difficult

to add new resources, since it requires adding new subtrees or moving existing subtrees. It is also difficult when patching together subtrees to balance demands on existing resources and to replicate for availability. These characteristics, when taken together, greatly complicate attempts to preserve an appearance of transparency.

Two other benefits of an independently managed Roe directory are host-independent support for heterogeneity (section 3.4.7) and added flexibility in integrating file and directory management algorithms. Examples here include replication (section 3.4.2), network modeling (section 3.4.4), initial placement (section 3.4.5) and migration (section 3.4.6).

3.4.3.1. The Structure of the Global Directory

The Roe global directory supports a UNIX-like hierarchical directory tree. We were motivated to adopt this organization because of its wide acceptance, ease of use, and the logical structure that it imposes on files. Roe allows users to choose arbitrary file and directory names. There is no encoding of location or host information in names visible to the Roe user.

The Roe directory references distributed files and is itself distributed. This raises issues that do not occur in centralized directories. Three key related issues are:

- The basis for partitioning the directory,
- Locating resources referenced through the directory, and
- Directory replication.

We will return to the issue of replication in the following section. The issues of partitioning and location have been dealt with in the past in a variety of radically different ways. We briefly surveyed work in distributed naming and directories in Chapter 2. As we described there, the technique of partitioning by subtree used by NFS, LOCUS, and other DFSs leads to unacceptable limitations on transparency.

R* [Lindsay 80] embeds the creation site in names and uses this information, along with forwarding addresses and cached hints, to access an object. Our desire to support transparency and

potentially frequent migration means that the approach used by R* is not directly applicable to Roe. However, the techniques of caching, forwarding addresses and unique name generation will be useful in the scheme we describe below.

Clearinghouse [Oppen 81] provides a hierarchical name space restricted to three levels. Names in Clearinghouse are logical (based on the application), rather than physical (based on server location), with each server containing complete information on where names in the current and parent nodes of the logical hierarchy may be resolved. This provides the network transparency that we desire, but the limited hierarchy is not useful in a distributed file system. Also, the widespread distribution of server location information complicates migrating servers. This is not an issue in Clearinghouse, where such migration is relatively infrequent, but it is an issue in Roe.

We would like the Roe directory to partition directory information logically along boundaries that reflect the way the information is used, that do not unduly restrict the location of this information, and that allow it to easily be distributed and migrated. Directories commonly support the following operations: read an entry (where 'entry' is the information describing a cataloged file) add an entry, delete an entry, update an entry and enumerate entries contained in a node of the directory tree. The 'enumerate' operation and the common practice of grouping related files in a directory lead us to make the unit of partitioning the set of entries in a directory node.

Resolving an absolute name (one that includes all components of the name) then involves starting at the root and looking up directories in turn to reach the one containing the needed entry. To minimize the overhead of this, Roe caches information on frequently used directories. This would typically include all directories from the root to the user's current working directory, but can also include paths to other directories. The resultant cached information forms a tree of directory information that may be used to avoid repeated lookups in frequently used directories.

There are two types of information in a Roe directory: 1) information on the directory node itself; and 2) information for each entry in the directory. The information on the directory node includes voting, version number, usage history and other information that will be described later. We do

not keep any information on the parent directory of a node. This is in contrast to UNIX and UNIX-like file systems, which include an explicit backpointer from a node to its parent. We have omitted the backpointer for several reasons. One is the complexity that it adds to migration. If backpointers are used and a directory is migrated, then children must be updated to reflect the new location. Backpointers are also used in UNIX to help preserve file system integrity. The Roe directory will be replicated and so necessarily updated atomically. This ensures that the directory will remain in a consistent state even in the presence of crashes, making backpointers for this purpose unnecessary. Finally, backpointers are used in UNIX to resolve relative references. Roe caches parent directory information for this purpose.

There are two basic types of directory entries in Roe: file entries and directory entries. In addition, there is a somewhat more complicated variant of a file entry, the *machine dependent file*, that will be described when we discuss heterogeneity (section 3.4.7). We also allow users to define their own entries. File and directory entries contain the same information:

- *name*: The user defined name of this entry.
- *type*: The type of this entry (file or directory).
- *UID*: The globally unique ID assigned to the file or directory when it was created. We use a combination of a host id and a sequence number guaranteed to be unique on that host.
- *location-hints*: Hints on where the copies of the file or directory are located.
- *r-hint*, *w-hint*, and *vote-hints*: Hints on the vote distribution and quorum characteristics for the file or directory.

Figure 3-1 shows an example of a Roe directory describing a replicated file. Note that the voting information in a directory entry is just a hint. The actual voting information, including the timestamp, is stored in the copies of the file itself. This allows a file to be referenced from multiple locations in the directory tree.

The "links" field in each file copy in Figure 3-1 indicates how many references to the file exist, and is used to ensure that a file with multiple names is not actually deleted until the last name is

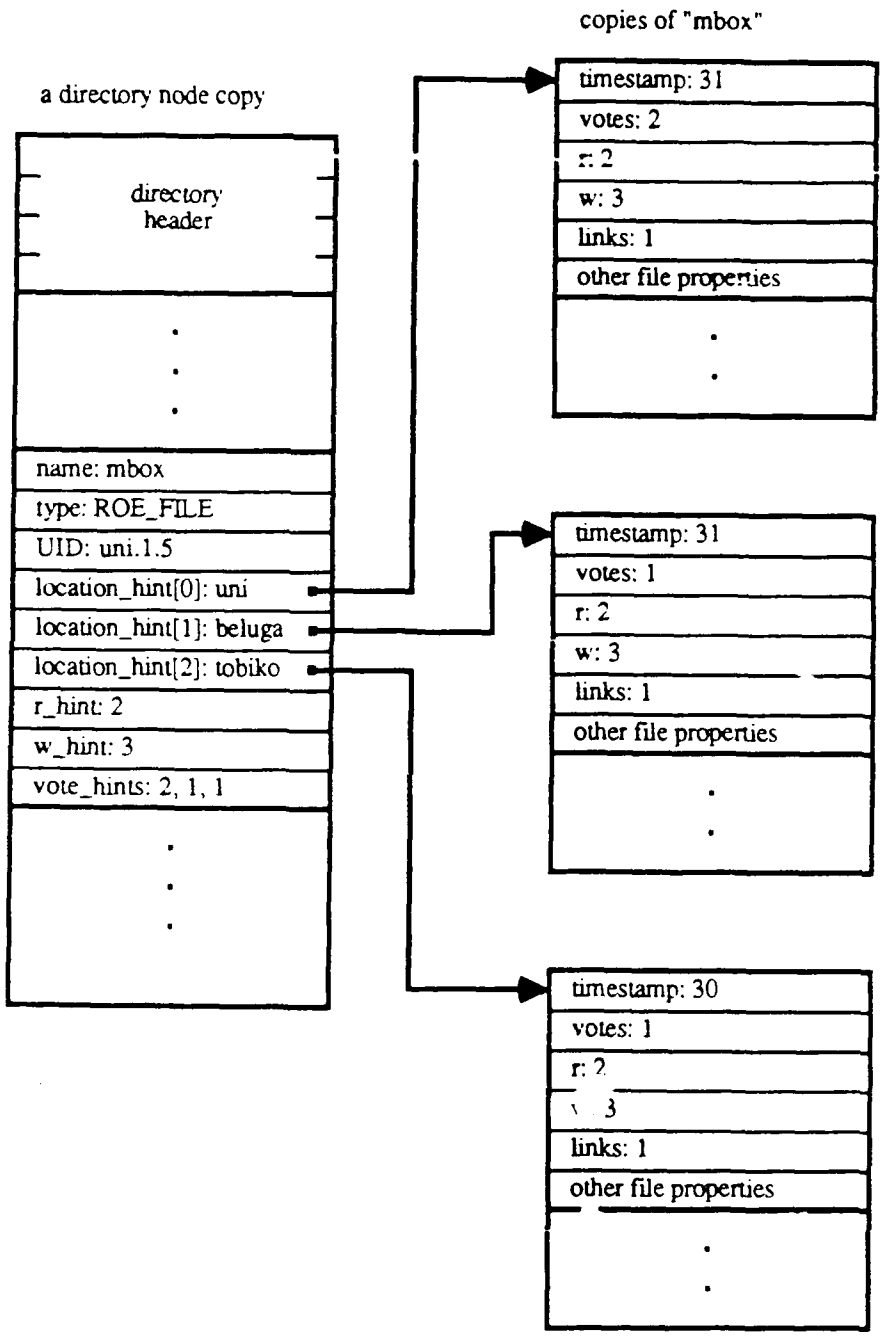


Figure 3-1: A Roe directory entry

deleted. Multiple links to directories are not allowed, since this would allow the formation of unreachable subgraphs. Garbage collection is one solution normally used to deal with this problem, but this is complicated in a truly distributed system (the description of a possible design for garbage collection in the Emerald system [Jul 88] is a good survey of the complexity involved in distributed garbage collection).

Voting and link information are actually *properties* of the file. Each file may have additional property information associated with it, as shown in the figure. This includes information useful to Roe (for example, usage information for migration) and user defined information. Properties will be described further when we discuss heterogeneity and Roe protocols.

The location information in a directory entry is also a hint. When a file is migrated it may not be possible to update the directory entry. Situations where this may arise and the techniques Roe uses to deal with it will be described when we discuss migration (section 3.4.6).

We keep voting hints in directory entries to aid in optimizing opens. An open for read, for example, only needs to contact enough copies to collect a read quorum. The voting hints can be used to limit the number of copies that are contacted when opening the file. The information here is regarded as *only* a hint to minimize the amount of information that must be kept current. The actual information is kept with the data to decentralize, as much as possible, the control of the data. If a voting is found to be incorrect because of changes in the file configuration, it can be updated with information gleaned from the open. Location hints are treated in a similar manner.

The preceding describes and motivates the basic structure of the Roe directory. There are a few other issues worth noting. These include finding the root of the directory during initialization, handling structural changes (such as directory renames) in the presence of caching, and caching file information.

Roe will generally run on most or all of the hosts on a network, with these hosts joining and leaving the network. When Roe is started on a host, it will need to open the root of the global directory tree as part of its initialization sequence. Since copies of the root directory may migrate, just

like any other directory, the new instantiation of Roe may not have up to date information on where the root is located. This information can be obtained by querying any running instantiation of Roe. In the unlikely case that there are no accessible instantiations, Roe uses information on the last known location of the root that it has saved in a local state file. This information may be slightly out of date, but should generally be current enough to allow the root to be opened. In the extremely unlikely case that this strategy also fails, the new instantiation of Roe can simply wait until an instantiation with more recent information on the root is started.

Renaming a directory can invalidate parts of the cached tree of information maintained by an instantiation of Roe. In order to provide users with a consistent view of the directory, we need to detect changes of this nature and flush or correct outdated information. This is done by having an instantiation that caches information on a directory register an interest with at least a read quorum of the copies. When structural changes are made, this registration information is used to notify affected parties. The same technique may be used to maintain a cache of information on frequently opened files.

3.4.3.2. Replicating the Global Directory

The availability, consistency, transparency, and reconfigurability requirements that we described earlier for files also apply to directories. Availability requirements lead us to insist on replicated directories. Consistency in this case includes seeing the effects of adding, deleting or modifying an entry when we subsequently do a lookup on the entry or enumerate a directory. Transparency and reconfigurability can be addressed, as with files, by migration. These considerations lead us to choose weighted voting as the basis for mutual consistency control in replicated directories.

Weighted voting, as described by Gifford [Gifford 79b], locks the entire object being accessed. While this is appropriate for files in our environment, it is not appropriate for directories. The specialized entry-oriented nature of operations on a directory, combined with the need for shared access, concurrency (especially at higher levels in the directory tree), and long term connections, makes whole-node locking both unnecessary and inadequate.

Variations on weighted voting that address these issues have generally done so by either reducing the granularity of locking or by using other concurrency control mechanisms. Bloch, Daniels, and Spector [Bloch 87] took the first approach. They associated a timestamp with each entry in a directory, and also with gaps between entries (to aid in collecting quorums after deletions). While this approach allows for higher concurrency, the large number of timestamps raises significantly the cost of connecting to a directory and verifying the currency of the directory as a whole. This is inappropriate in an environment where connections to directories are frequently made and broken. It complicates caching as well. Herlihy [Herlihy 84] proposed a variation of weighted voting that performed concurrency control based on timestamps and relations between operations on abstract types. While his work provides a powerful set of general techniques for increasing concurrency in replicated objects, the complexity of log-based concurrency control mechanisms are difficult to justify in our environment, particularly given its heterogeneous nature. They also do not address the need for maintaining low cost connections.

The following variant of weighted voting provides the concurrency control we need, with support for caching and long-term connections that don't lock out other users. While it does not have the generality of the other approaches and limits concurrency for writers (neither of which is expected to be an issue in our environment), it is easy to implement, inexpensive, and meets the needs of Roe.

The algorithms is as follows: When an instantiation of Roe connects to a directory, at least a read quorum of votes is collected from the node copies. At this time, the user of the directory is *registered* with these copies. Registration differs from holding a lock in that the registration may be 'broken', with notification, as explained below. From the read quorum, a current copy is selected and read requests are directed to it.

Writing requires that updates be made atomically to current copies containing at least a write quorum of votes (currency is verified during read quorum collection). We send to all current copies the update requests (to add, delete or modify an entry). If they are willing to make the change they respond with an acknowledgement. If a write quorum is collected, then the user can

instruct the servers to commit. At this point, the changes are actually made (become permanent and visible to readers) and the timestamps of the participating copies incremented. If a write quorum cannot be collected, the request is aborted and, depending on the error, may be retried. To preserve the meaning of the version number, only one write may be active (in the process of collecting votes or committing) for a directory node at a time.

An operation, such as modifying an entry, that depends on the previous information needs a bit of special handling to guard against making changes based on invalid data. We send, in the request to modify an entry, both the new information and the data in the entry upon which the changes are based. If these data do not agree with the information currently in the directory, the request is rejected. Since an individual directory entry is small, little extra overhead is involved in doing this.

The registration information is used when a writer finds that he cannot update all current copies of a node, even though he is able to collect a write quorum. In this case, the writer, in the commit message, tells the participating directory copies to notify registered readers that they may no longer be reading a current copy. Since the sum of read and write quorums is greater than the total number of available votes, there is always some overlap between the quorums and so all readers will be notified. Readers also receive notification when copies they have registered with become inaccessible due to partitioning or node crashes. This is handled by maintaining IPC connections to these copies and using the automatic failure detection facilities of the underlying IPC [Moore 82]. The combination of these two notifications ensures that readers will be reliably informed of problems.

The registration mechanism can also be used to handle directory cache invalidation due to renames or other updates. In this case users would register interest in node modifications and be notified when significant changes are made.

3.4.4. The Network Model

Intelligently placing and migrating files and directories requires current information on the underlying hosts and networks. Roe maintains an abstract *network model* that contains this information. The Roe design we are presenting here attempts to place and migrate to minimize delay and maximize availability, consistent with the overall goal of transparency. The information maintained in the network model we will be describing reflects this. Information in the network model is also used to support heterogeneity, and to allow Roe to take advantage of the varying capabilities of hosts in a heterogeneous network.

The network model contains both static and dynamic information on hosts and networks. Each host with an instantiation of Roe maintains its own network model of hosts and network fragments that it is actively using. To aid in decentralized decision making and to simplify interpretation, this information is expressed relative to the host maintaining the model. Only partial information is kept to avoid scaling problems that would result if an attempt were made to maintain a current model of the entire network.

A network model contains 3 elements: hosts, network switching elements (gateways), and the network itself. The model maintains at least the following information on each host:

- *Availability*: A static estimate of the fraction of time the host can be expected to be available.
- *Hardware base*: The type of host hardware.
- *Host operating system*: This, combined with the hardware base, can be used to make machine-dependent decisions.
- *Free space*: A dynamic estimate of the amount of free space currently available for use by Roe on the host.
- *Delay*: A dynamic estimate of the delay involved in performing an operation. This includes network and switching delays from the current host.
- *State*: UP if the host is known to be currently accessible, DOWN if it is known to be currently unaccessible, and UNKNOWN otherwise.

This can, of course, be extended as necessary to meet the needs of new placement or migration algorithms. For each network segment and gateway, at least the following information is maintained:

- *Availability*: A static estimate of the fraction of time a network segment or gateway can be expected to be available.
- *Bandwidth*: A measure of the rate at which data can be transferred by the network or gateway.
- *Delay*: A dynamic estimate of the delay to be expected in passing through a gateway or network segment.
- *State*: UP, DOWN or UNKNOWN.

The model also encodes information on the network topology that allows the various components to be pieced together. Figure 3-2 shows an example of a network model for two bus networks connected by a gateway.

Information contained in the network model comes from three sources: from a static network state file maintained by Roe administrators, from the hosts being modeled, and from dynamic measurements. The network state file contains information on well-known hosts supporting Roe and on the network itself. An instantiation of Roe on a host can be asked to return information on the host on which it is running. This includes both static information such as the host operating system, and dynamic information on free space. Delay can be calculated either by direct measurement of the time required for operations or by combining estimates from network and host components. Hosts are added to the network model as they are accessed (for example, when a directory is opened) and can be deleted if the model grows too large or if they haven't been accessed recently.

Using the information in the network model, it is meaningful to talk about such things as the distance from a host to a file copy (in terms of delay) and to estimate the likelihood of that copy being available. These figures can be used in migrating copies and placing new ones. They may also be used in making decisions on which copy of a file or directory to access when performing

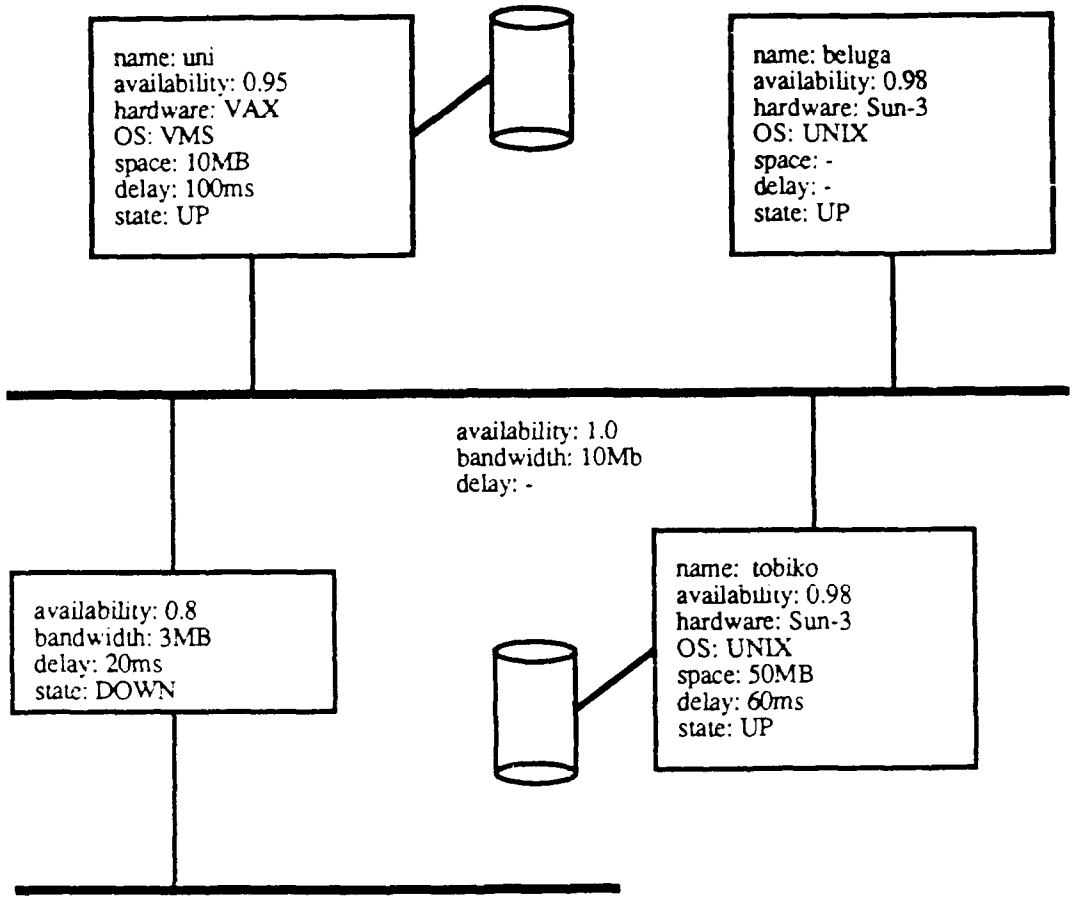


Figure 3-2: The network model

operations.

3.4.5. Initial Placement

When Roe creates a new file or directory, a decision must be made on how many copies to create and where they are to be placed. Users will generally not know, or wish to know, what resources are available for storing new information. Instead, the replication and placement of files and

directories is managed by Roe to preserve network transparency. Varying the placement of files can have a dramatic effect on performance. Studies of some versions of the Andrew file system [Morris 86] have shown as much as a 5 to 1 difference in utilization between file servers, which results in dramatic differences in response time under load. It is important that Roe takes into account factors such as this when placing files.

Most of the previous work on initial placement that we described earlier (section 2.7) was concerned with finding optimal solutions that minimized various cost functions. This previous work was motivated by the need to carefully place portions of a DBMS, where there is a large potential benefit to minimizing access costs. Optimal file placement is NP-complete, even without considering resource limitations. In this case, the cost of solving the NP-complete placement problem is justified.

In our case, optimal solutions are not appropriate for several reasons. The interactive nature of Roe makes it important that file placement decisions be made quickly, and so algorithms that find optimal solutions slowly are unacceptable. We expect that issues such as congestion, which depend on the interaction of accesses to many files, will play an important part in determining expected delays, and hence placements. This further complicates attempts to find optimal placements. We will also, in general, have only partial knowledge of the network that Roe is running on, and limited or no knowledge of the activities of other users. Finally, Roe has the ability to migrate files based on usage information that isn't available when the file is initially placed, so mistakes do not necessarily have a permanent impact.

For these reasons, Roe is designed to make use of placement heuristics. There has been some previous work in placement heuristics that is relevant to the goals of Roe. Bannister and Trivedi suggested making new assignments to the most lightly loaded servers to minimize delay [Bannister 82]. Barbara and Garcia-Molina presented heuristics for vote assignment to maximize availability [Barbara 86]. Factors present in Roe that complicate the use of these and other algorithms include the heterogeneity of the environment (both in terms of performance and machine types), the interaction of placement with migration, resource limitations, and our desire for algorithms that

exhibit decentralized control and are able to operate with partial knowledge.

The use of decentralized control in Roe means that each instantiation may, if it wishes, use a different placement heuristic, depending on its needs. The primary objective of file placement in Roe, based on the goals we discussed earlier, is to attempt to minimize access delays while insuring high availability, subject to the other constraints we have described. This can be done, in the architecture we have been describing and using the information provided by the network model, as follows: When a file is created a replication factor is selected. This will, by default, be based on the replication factor of the directory where the file will be cataloged. The creator of the file may specify additional information on the type of the file (temporary or permanent) and important characteristics of the file (high availability vs. high performance, usually read vs. usually written, and so on) as part of creation. These are used to modify the default replication factor and to assign tentative vote quorums. For example, files that are marked as being temporary are normally only given one copy. Files where high availability is requested are given a higher replication factor.

Each copy of the potentially replicated file is then independently placed. This is done using state, free space, delay, and availability information in the network model maintained by this instantiation of Roe. By default, hosts that currently have the lowest delay are chosen, but only if their availability exceeds a threshold and if sufficient space is available. If a machine-dependent file is being created, there is a further restriction that the new host be of the same type as the one creating the file. If the creator of the file has asked that placement emphasize availability, this will be given primary consideration, with delay being secondary. In this case an attempt is also made to locate copies on hosts that currently support the directory, as this will tend to decrease failure points and so increase availability.

While the algorithm we have described here is by no means optimal, it does provide a low cost approach to automatically placing files. It takes advantage of the transparency provided by the Roe design to dynamically balance load among active servers, to increase performance by selecting lightly loaded and higher performance servers, to increase availability through replication and

the use of reliable servers, and it takes into account constraints imposed by limited resources and heterogeneity.

3.4.6. Migration

The conditions that determined the initial placement of file and directory copies change over time. For example, congestion makes some servers less attractive than they originally were. Users change locations. Usage patterns of a file change and a history of usage provides more accurate information than that used for initial placement. File creation and deletion change the available space on a device. Faster or more reliable servers become available. If the changes are significant, it may be worthwhile to move, or *migrate*, copies to adjust to the changes. The benefits of migration can be significant. Using migration in the Emerald system [Jul 88] resulted in a 22% decrease in execution time relative to the execution time without migration.

This section describes the architectural support that Roe provides for migration. We also outline here in general terms migration strategies that Roe supports, but defer discussion of actual migration algorithms until Chapter 7. We expect migration to be sensitive to file and directory reference patterns, and so algorithms are best discussed in that context.

Most of the previous work in file migration that we described in Chapter 2 focussed on long-term archival migration. That work showed that file size, type, age, and time since last reference were all useful predictors of the time until next reference, and hence could be used as a basis for archival migration. While archival migration is one strategy that Roe supports, the relatively small time delay and high bandwidth of LANs make migration on a much smaller time scale feasible. The temporal locality that we expect to see in file reference patterns also argues in favor of short-term migration strategies. The Emerald system took advantage of this temporal locality by migrating objects to the user when they were first referenced and saw substantial performance improvements. Work on optimal and heuristic migration algorithms by Sheng [Sheng 86] also demonstrated the advantages of short-term migration. Even with an optimal initial file assignment, she was able to achieve a 10% decrease in storage, communication, and I/O costs using a migration

heuristic that estimated future usage based on recent access history.

Our needs and constraints are somewhat different than that of earlier work. As with file placement, we will be migrating to increase availability and decrease delay, subject to processor, space, and heterogeneity constraints. The arguments for using heuristic algorithms able to operate in a decentralized fashion and with partial (possibly incorrect) knowledge that we used for replication and initial placement also apply to migration. Hence our concern here is in providing support for heuristics to meet these goals.

Migration algorithms can be divided into three general categories: demand-based, anticipatory and compensatory. *Demand-based migration* is done in response to an explicit request for a resource. Emerald support for migration on first reference is an example of demand-based migration. *Anticipatory migration* is done based on expected usage. If resources are typically used together (a trivial example is a directory and files that it references), migrating one to meet a demand may trigger further migration in anticipation of further references. *Compensatory migration* is done to compensate for changes in the system that have taken place since initial file placement was done. Examples here include migrating files away from heavily used servers to ease congestion (and decrease delay), migrating to new servers to balance load, and grouping together directories and the files they reference to increase availability. This type of migration is perhaps best thought of as an on going background activity.

Making intelligent migration decisions requires an estimate of future accesses that will be made to a file or directory. Previous work has suggested that future accesses may be estimated using past access history. Roe keeps with each file and directory, as a property, information on recent accesses. For each access Roe records the user, what host it was from, when it was made, the type of access, and the percentage of the file read or written. File size, type, and age, other predictors of future usage, are also available as properties of files and directories. In addition, information on the state of the network is required. This is particularly important for compensatory migration. Network information requirements for migration are similar to those for initial placement. The relevant network information is available from the network model.

From the point of view of the lower levels of Roe, there are two types of migration: migration by name and migration by internal identifier (UID). Migration by name is typically done in response to an explicit user request of some sort (demand or anticipatory migration), or to group together related objects in the naming tree (compensatory migration). Migration to relieve congestion or free up space on a server (compensatory migration) is more easily done by UID, based on files or directories on the affected server.

Migration by name is done by first opening and locking the copy to be migrated. This ensures that the copy will not be able to vote multiple times while it is being moved (otherwise it might be possible for the old copy to participate in one quorum and the new one in another, resulting in inconsistencies). The old copy is also flagged for deletion at this time. A copy is tentatively made on the destination host, and the directory is updated to indicate the new location. These changes are then atomically commited.

If a file is referenced by more than one directory, then updating the directory the file is referenced through is not enough. The other directories will still contain obsolete information on the old location of the copy. In this case a *forwarding address* is left behind on the old host. The forwarding address contains the new address of the copy and a count of the number of directories containing obsolete information (just one less than the link field from the file). Unlike R*, where forwarding addresses exist for the lifetime of the object, forwarding addresses in Roe can be discarded after all directories containing obsolete information are updated. When a file copy is referenced from an out of date directory and a forwarding address is encountered, the directory is updated and the reference count in the forwarding address is decremented. When it falls to 0, the forwarding address can be deleted. If a file referenced by multiple directories moves frequently, resulting in a chain of forwarding addresses, path compression techniques [Fowler 85] may be used to eliminate unneeded links in the chain.

It should be noted that this technique only works because the algorithms used by Roe assume that information contained in directories is only a hint. Treating this information as a hint places actual control of files with the files themselves and minimizes the amount of work that is required

for migration.

Migration by UID proceeds as with migration by name, except that it isn't possible to update the directory. In the case of migration by UID, a forwarding address with a reference count equal to the link count of the copy is left behind. Directories are updated the next time they are used to reference the copy.

3.4.7. Support for Heterogeneity

Roe is intended for use in a heterogeneous environment. The support Roe provides for heterogeneity may be grouped into the following areas:

- The network model.
- Automatic conversion of basic types.
- Machine-dependent files.
- Properties.
- Globally consistent naming and interfaces.

The network model, as we described earlier, includes information on the hardware and software base, and on delay, availability, and free space for each host it describes. The information on hardware and software base is used in making decisions that depend on this base. An example is the selection of a quorum in a machine-dependent file (described below). The remaining information provides a representation of the relevant properties of hosts that allows hosts of dissimilar performance, capacity, and reliabilities to be integrated, while still making effective use of the strengths of each host.

The Roe design provides for the automatic conversion of basic data types between hosts. This can either be done automatically by the IPC mechanism (as it is done in Rochester's IPC [Moore 82]) or explicitly by conversion to and from some canonical form (see, for example, Sun's XDR [Lyon 85] or Cronus's Cantypes [Dean 87]).

Not all files can or should be converted when moving between hosts. Executable files containing machine code, for example, are only useful on machines of certain types running a particular operating system. To retain transparency and allow the Roe naming tree to be shared between hosts of dissimilar types, we introduce a new file type, the *machine-dependent file*. A machine-dependent file appears to the casual user to be a single Roe file, but it contains separate file suites for each distinct machine/operating system combination on which it may run. When the file is opened, Roe chooses the correct file suite using information on the user's host as contained in the network model. Figure 3-3 shows an example of a directory entry for a machine-dependent file.

Roe associates a *property list* with each file and directory. A property list is just a list of name/value pairs. They were included in the Alto file system [Thacker 79] to describe, among other things, the data type of a file. They perform a similar function in Roe, but may also specify arbitrary user and Roe system information. For example, voting and usage information for a file or directory copy is stored as properties of the copy. Information such as file size and last modification date is also represented as file properties, with Roe converting between property and actual system representation as necessary.

In addition, information that is specific to particular host operating systems (for example, record structure) is stored as file properties. This provides an unobtrusive method for accommodating the needs of a wide range of heterogeneous systems.

Properties may also be specified in file creation and other operations. They provide a means of allowing the user to specify optional advisory information to Roe. Examples here include file type, desired availability, and performance characteristics.

Finally, we have developed an architecture for Roe that defines a common host-independent interface between local hosts and those portions of Roe that interface with many hosts. We have also defined a file access protocol that supports transparent access to Roe files and directories. This access protocol has been designed to be flexible enough to support higher-level translation of host specific file operations to operations on Roe files for a wide variety of operating systems. The

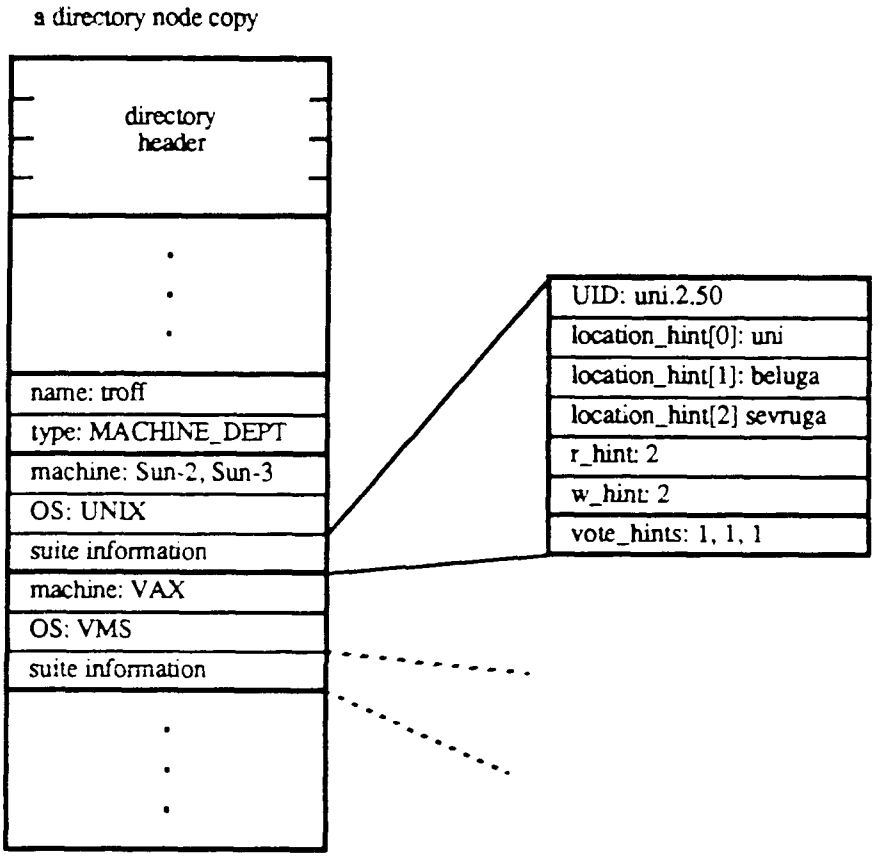


Figure 3-3: A machine-dependent file entry

Roe architecture and protocols are the subject of the next section.

3.5. The Architecture of Roe

The previous section presented the general approach used by Roe to provide transparency. In this section we outline an architecture that may be used to implement the Roe approach. Our purpose here is to define a framework that will allow Roe to be easily implemented in a heterogeneous

environment. Our general approach has been to define common host-independent interfaces for components that provide access to resources on a host, and to structure higher levels of Roe in a way that supports distribution and replication of functionality and allows decentralized access to Roe resources. The architecture we describe here should be thought of as a logical one, not necessarily a physical one.

Section 3.5.1 describes the purpose and distribution of Roe components in our architecture. Sections 3.5.2 and 3.5.3 outline protocols that define the interfaces between these components. The material presented here forms the basis for the prototype implementation that we will be describing in Chapter 4.

3.5.1. Organization and Distribution

Roe may be logically broken down into five different types of components. Three of these components provide access to local information (files, directories, and host status), one manages transactions on replicated resources, and one performs the work necessary for name translation and opens. The local components provide a uniform, host-independent interface to the two higher level components. The two higher level components piece together the local resources, using the techniques we outlined in section 3.4, to give users a transparent, globally consistent view.

The five types of components, and their purposes, are:

- *Local File Servers*: provide access to individual copies of files located on a host.
- *Local Directory Servers*: provide access to individual copies of directory nodes on a host.
- *Local Representatives*: provide higher levels with information on the status of the host (hardware and software base, space available and so on) and spawns new servers on request.
- *Transaction Coordinators*: distribute user requests to replicated copies of files, coordinate commits, and mask single copy failures.

- *Global Directory Servers*: accept user requests to open files or access directories and map the requests into operations on a set of file or directory copies. Global Directory Servers also perform initial placement and migration for files and directories.

A *Local File Server* (LFS) provides access to file copies located on a host. There is one local file server for each host that provides file storage. An LFS supports an abstraction of a file as a stream of data, with an associated list of properties. Applications may use property lists to impose more structure on the data. The LFS on a machine manages file space used by Roe on the host, provides atomic access to data and properties in files (including voting information), and implements the copy-level locking needed by the weighted voting algorithm.

A *Local Directory Server* (LDS) mediates access from Global Directory Servers to copies of directory nodes on a host. An LDS supports the abstraction of a directory as a collection of entries (name, quorum information pairs) plus a list of properties. It services requests to read, update, delete, add, and enumerate entries in a node copy, provides access to properties, maintains voting information for the directory, and ensures that updates are performed atomically. There is one Local Directory Server on each machine that has directory information.

A *Local Representative* (LR) provides Global Directory Servers with information on the status of the host (hardware and software base, space available, and so on) and spawns new servers on request. There is at most one local representative per host.

A *Transaction Coordinator* (TC) distributes updates to replicated copies of files, coordinates commits, and masks single copy failures. It is the TC that handles user operations on open files. It directs reads to a current file copy near the user, switching to a new copy if the one in use fails and a quorum is still held. It transforms a write operation into updates on all open copies of a file. When a file is closed it ensures that updates are atomically applied. There is one transaction coordinator per active user.

A *Global Directory Server* (GDS) accepts user requests to open files or access directories and maps the requests into opens on a quorum of file copies or operations on a quorum of directory

copies. GDSs maintain the hierarchical Roe name space and use it in resolving requests on files and directories. Each GDS also maintains a network model and uses it to perform initial placement and migration of files and directories, and to place the transaction coordinator that will handle user operations on open files. The number of global directory servers depends on the network load and configuration.

Figure 3-4 shows an example of how the components interact in opening and using a file (in this case, an unreplicated one). The user sends the initial request (a) to the GDS. The GDS makes use of cached directory copies and, if necessary, contacts LDSs to read entries necessary to reach the file (b). The GDS then opens the file with the requested mode (c), using the weighted voting algorithm to collect and verify a quorum if the file is replicated. The resultant quorum is passed (d) to the TC, which accepts user requests (e), and passes them on to one or more selected file copies (f). The protocols that are used between the various components are described in the next two sections.

The actual amount of interprocess communication required here depends on the relative locations of the user and Roe components involved, and on the degree of integration of the components. An implementation of Roe that was concerned with performance would tightly bind together the components on a host to decrease communication overhead.

In an environment where it is important to be able to gain access to network resources with minimal implementation effort (often the case in a rapidly evolving heterogeneous environment), a new host could choose to implement a subset of Roe components and rely on other hosts in the network for the rest of the services. Figure 3-5 shows an example where two hosts (beluga and sevruga) support all of the Roe components, but another, uni, only provides storage for files. The host tobiko just implements the common IPC used by Roe components. This gives users on tobiko access to Roe through components on beluga and sevruga. Performance may suffer, since all access is remote, but functionality does not. Implementing portions of Roe on tobiko at a later time would result in transparent performance improvements for users, since it would then be possible for Roe to migrate information to this host and so avoid network penalties. This incremental

buy-in strategy encourages heterogeneity by allowing new types of hosts to be easily integrated.

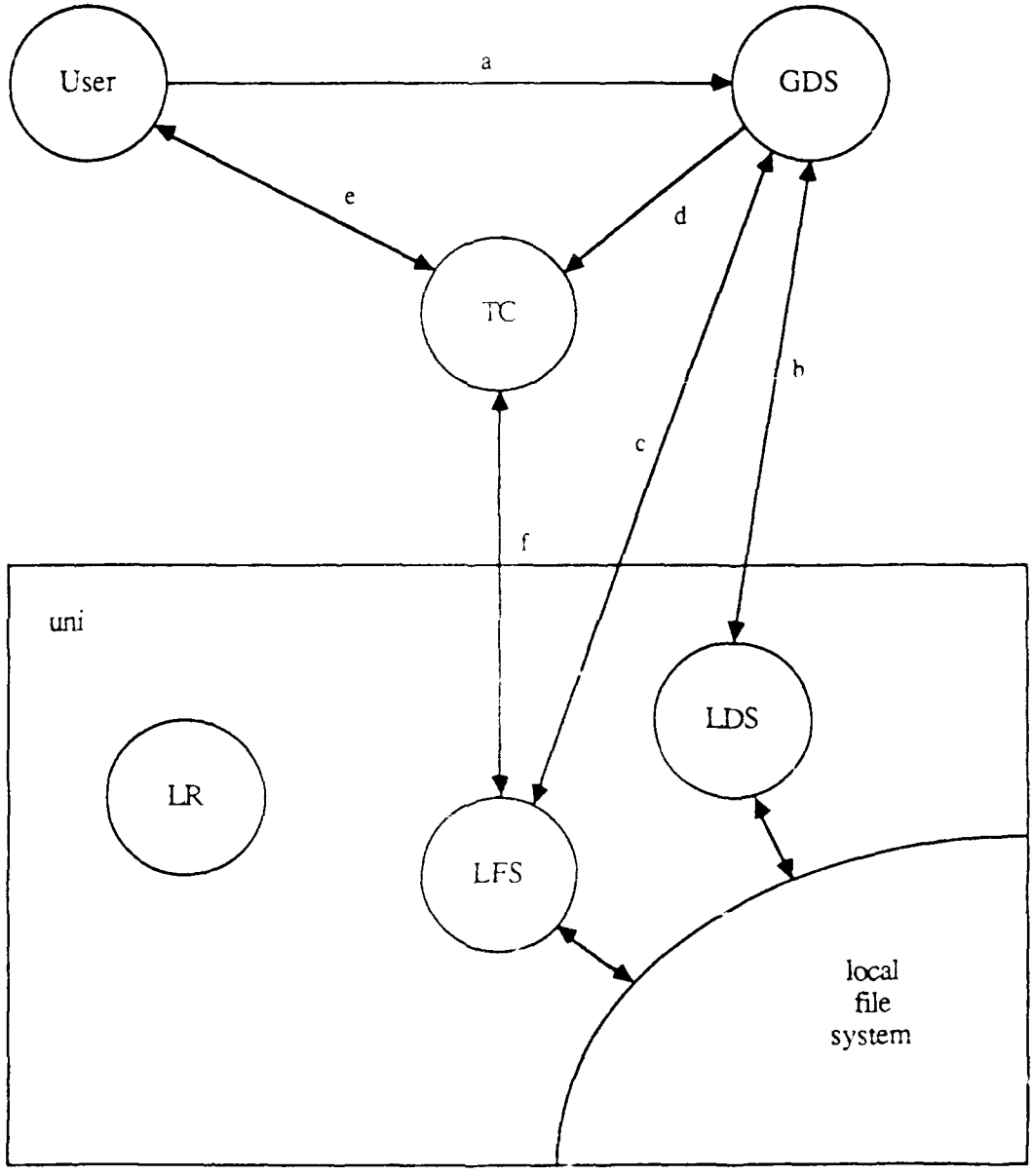


Figure 3-4: Opening and reading a file

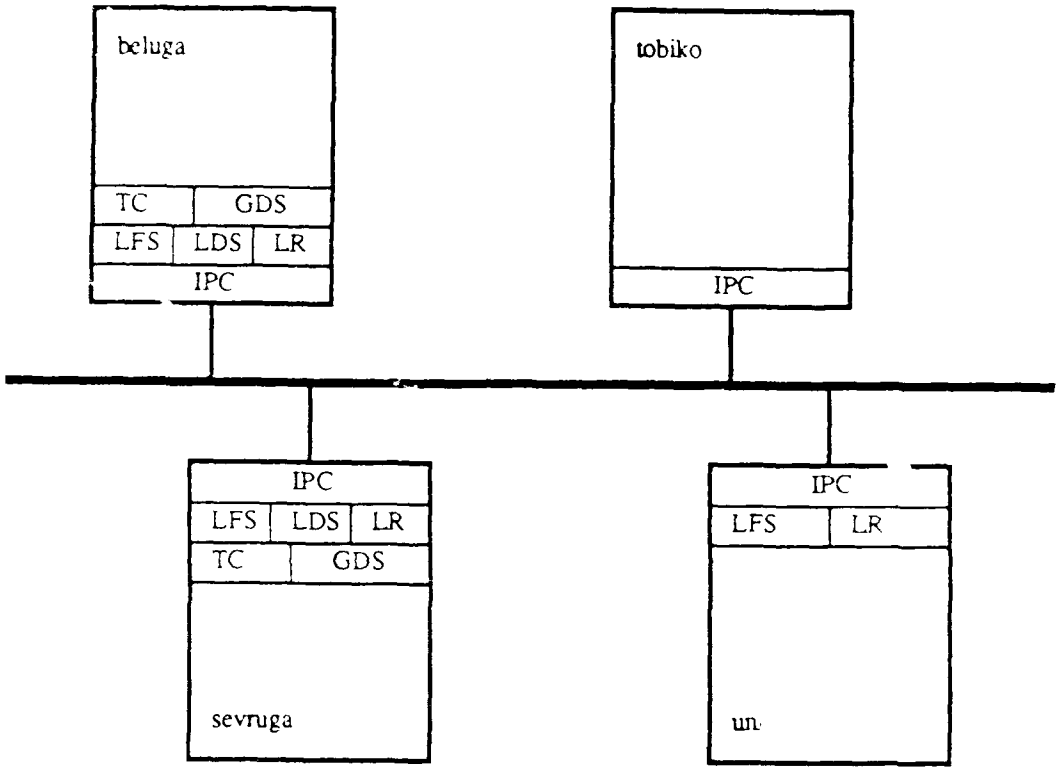


Figure 3-5: Roe component location

3.5.2. User Protocol

This section defines the protocol used to access Roe. The Roe file access protocol is distinguished by its support for properties, atomicity, and distribution.

Properties allow both users and Roe itself to associate information with files that aids in interpreting data. Properties are used by Roe to store information needed for voting, data conversion (to support heterogeneity), and migration. They are also used to specify special characteristics of files used in initial placement and migration, and may be used to set and retrieve information about files (for example, size and modification dates). As such, they provide a method for extending the Roe protocol, and for adapting it to the needs of other systems. Given the ability to associate properties with a file, mapping host file access calls into the Roe file access protocol is generally straightforward. This can be used to make Roe appear to be an extension of a host's file system.

Atomicity allows users of Roe to guarantee the state of files at any given time. This predictability simplifies the implementation of higher level applications that use Roe.

The protocol has been designed with a message-based distributed environment in mind. The protocol itself is message-oriented, and it includes support for streaming of data, and third party transfers.

The protocol is oriented around the notion of port types. A *port* is any unique identifier that provides a way to address a potentially remote component of Roe. There are 3 types of ports in the user protocol, each of which accepts a different class of messages and serves a different purpose. *Packports* provide access to the file system. *Directoryports* support commands creating and reading directory entries, opening files, and so on. *Fileports* are used for operations on an individual open file. Each operation given below is specified using the format "OPERATION (arguments) → reply." Optional arguments are enclosed in brackets. The source and destination ports are implicit in the description that follows.

There is one message to packports:

LOGIN (username, password) → directoryport

A user must be logged in to the Roe system in order to issue requests. A LOGIN message containing information necessary for authentication is sent to a packport. The server

checks the validity of the user and establishes the user's working directory. The reply contains a directoryport to which the user sends subsequent directory operations.

The following messages are sent to directoryports:

CH_DIR (path) → success

Changes the default directory used for interpreting operations on this directoryport.

CREATE_DIR (path [, properties]) → success

Creates a directory node with the specified path name. The optional properties list may be used to specify the performance and availability characteristics of the new directory and to initialize properties in the directory.

CREATE_ENTRY (path, value) → success

Creates a directory entry with the given path and value. This allows objects other than files to be stored in the Roe global directory.

DELETE (path) → success

Deletes the file, directory, or entry with the given path.

LOGOUT ( ) → success

This ends the server-client relationship.

OPEN (access, file path [,properties]) → fileport

Opens a file, returning a fileport that may be used for file operations. "Access" describes the mode of the file open (read, write, read/write, create if necessary). The optional properties may be used when creating a file that does not already exist to specify availability

and other properties of the file and to initialize its property list. This operation is successful if the appropriate quorum of copies can be gathered.

READ_DIR (destination port [, pattern]) → number of items read

Returns a list of files in the working directory, optionally filtered through a pattern matcher. The file names are sent to "destination port" in a WRITE operation.

READ_ENTRY (path) → value

Returns the value associated with given user-defined entry.

READ_PROPERTIES (path [, property names]) → property list

Returns properties associated with the named object. Optionally, return the properties with the given names.

RENAME (old path, new path) → success

Changes the name of a file, directory, or user entry.

WRITE_PROPERTIES (path, property list changes) → number of properties changed

This operation modifies the object's properties according to the changes given. Users may only modify user-defined properties.

The remaining messages are sent to fileports:

ABORT ( ) → success

Undoes all operations since the last commit on this file.

CLOSE ( ) → success

Closes the file and deallocate its fileport. If the file was opened with atomic access, any uncommitted operations are aborted.

COMMIT ( ) → success

Makes changes since the last COMMIT or OPEN permanent. This is the second phase of the two-phase commit protocol.

READ (destination port, number of items to read, number of items per block) → number of items read

This message requests that some number of items from the file be written (using the fileport WRITE operation with the given blocking factor) to the specified destination port. Reading begins at the current position of the read-write pointer; when the operation is done the read-write pointer is positioned after the last datum read.

READ_PROPERTIES ([property names]) → property list

Return the file's property list. Optionally, returns the properties with the given names.

SEEK (position) → success

Changes the value of the read-write pointer to "position."

SYNC ( ) → success

The first phase of the two-phase commit protocol. A success reply indicates a willingness and ability to commit.

TELL ( ) → position

This operation returns the current value of the read-write pointer.

{ WRITE (data) }* WRITE_ACK (data) → number of items written

The WRITE operation is special in that it is composed of multiple messages. The result returned is the total number of items (from the preceding stream of WRITE messages and the final WRITEACK message) that were successfully written.

WRITE_PROPERTIES (property list changes) → number of properties changed

This operation modifies the file's properties according to the changes given. Users may only modify user-defined properties.

3.5.3. Internal Protocols

This section outlines the protocols recognized by the various components of Roe. The intent of this section is to define the interfaces between the components, and to characterize the abstraction supported by each type of component.

Local File Server Protocol

The local file server recognizes the user fileport protocol described in the previous section. In addition, it implements the following operations used by GDSs:

ENUMERATE (destination port [, properties]) → number of UIDs returned

Returns the UIDs of file copies currently maintained by the LFS. Optionally, only returns UIDs matching the specified properties. The UIDs are sent to "destination port" in a WRITE operation. This operation is intended for use with migration algorithms that work at the host level.

INQUIRE (access, UID [, properties]) → fileport, quorum information

Used by a GDS to open a file copy. It differs from OPEN in that it specifies the internal

UID used by Roe to identify objects (and recognized by the LFS) and returns the quorum information contained in the file copy.

LOGIN ("roe", roe password) → LFSport

Authenticates a GDS and returns to it a port for future operations.

LOGOUT ( ) → success

Used by a GDS to terminate its connection to the LFS.

The LFS attaches special significance to some file properties. For example, the type of data in the file is a property, and is used to type messages containing the data. Another property specifies the "state" of the file, and is used by the GDS for synchronization during migration. An example of this is given at the end of the section.

Local Directory Server Protocol

A GDS authenticates itself with an LDS using the standard LOGIN protocol. This operation returns an *LDSport*, which may be used to manipulate complete directory nodes. LDSports support the following operations:

CREATE_NODE (UID) → success

Creates an new, initially empty, directory node on this host with the given UID.

DELETE_NODE (UID) → success

Deletes the empty directory node with the given UID.

ENUMERATE (destination port [, properties]) → number of UIDs returned

Returns the UIDs of directory nodes currently maintained by this LDS. Optionally, only returns UIDs matching the specified properties. The UIDs are sent to "destination port" in

a WRITE operation. This operation is intended for use with migration algorithms that work at the host level.

OPEN_NODE (UID [, register]) → nodeport, node state, quorum information

Opens the directory node with the given UID, returning a nodeport for it, along with quorum information and information about the node state. "Node state" specifies whether or not the node is currently in use and up to date. Version numbers are ignored for nodes that are in use (since ongoing transactions may update them while votes are being collected). If "register" is specified, the GDS will be notified of changes in the directory or its state.

LOGOUT ( ) → success

Used by a GDS to terminate its connection to an LDS.

READ_PROPERTIES (UID [, property names]) → property list

Returns properties associated with the named directory node. Optionally, return the properties with the given names.

WRITE_PROPERTIES (UID, property list changes) → number of properties changed

This operation modifies a directory node's properties according to the changes given.

Nodeports are used primarily to access entries inside nodes. *Nodeports support the following* operations:

ADD_ENTRY (entry [, transaction class]) → success

Adds a new entry to the directory node. By default, node updates such as ADD_ENTRY are applied when received. If an entry addition, deletion, or modification is part of a transaction, "transaction class" will be present. If it has a value of ENDING, an implicit

SYNC is done for this node before returning. This avoids the need for an explicit SYNC which, given the small size of directory entries, would be a significant additional overhead.

CLOSE ( ) → success

Closes an open directory node.

DELETE_ENTRY (entry name [, transaction class]) → success

Removes the entry with the given name from the directory.

ENUMERATE_ENTRIES (destination port, {ALL | JUST_NAMES} [, pattern]) → number of entries matched

Returns all entries in the node (if "ALL") or the names of all entries in the node matching "pattern." The UIDs are sent to "destination port" in a WRITE operation.

READ_ENTRY (entryname) → entry

Returns the contents of the entry with the given name.

READ_PROPERTIES ([, property names]) → property list

Returns properties associated with the directory node. Optionally, returns the properties with the given names.

UPDATE_ENTRY (new entry [, transaction class] [,old entry]) → success

Changes an existing entry. If included, "old entry" must match the current value of the entry.

WRITE_PROPERTIES (property list changes) → number of properties changed

Modifies a directory node's properties according to the changes given.

In addition, nodeports recognize the ABORT and COMMIT operations used for two-phase commit (SYNC is implicit) and the WRITE operation defined in the user protocol. The WRITE operation is used in conjunction with READ_PROPERTIES, WRITE_PROPERTIES, and ENUMERATE_ENTRIES to migrate directory copies and to bring up to date obsolete copies. Copies are frozen (by setting a special state property in the copies) for the brief period of time that is required to do this.

Local Representative Protocol

The LR supports the following operations:

LOGIN ("roe", roe password) → LRport

Authenticates a GDS and returns to it a port for future operations.

LOGOUT ( ) → success

Used by a GDS to terminate its connection to the LR.

STATUS( ) → properties of this host

Returns to the GDS information on the local host. This information is returned in the form of a property list to allow it to be easily manipulated and expanded.

SPAWN( ) → TCport

Creates a new Transaction Coordinator on this host and returns a port to it.

Transaction Coordinator Protocol

The TC supports the user fileport protocol defined in the previous section and translates fileport operations into operations on multiple fileports. In addition, it accepts the following operation from the GDS:

FILE_SUITE (userport, access, r, w, votes, LFSports [, entryport]) → fileport

Used by the GDS to supply a TC with a description of a newly opened file. "Userport" identifies the user and allows the TC to return a fileport to the user. "Access" is from the OPEN. "R", "w", "votes", and "LFSports" describe the file quorum. "Entryport" is present if there is a directory change that should be committed as part of file commit.

Global Directory Server Protocol

The GDS supports the user packport and directoryport protocols defined in the previous section. In addition, it accepts two-phase commit operations (ABORT, COMMIT, and SYNC) from TCs for use in synchronizing entry changes with file commits, and WRITE operations from LDSs and LFSs as part of retrieving information on file and directory copies on a host.

Finally, changes in directory nodes that the GDS has registered with are sent using the following operation:

NOTIFY (condition, description)

Used by an LDS to notify a GDS of changes in the state of a directory copy. This can be generated in response to directory updates (in which case "condition" contains the operation and "description" the new entry), or to notify a GDS of update failures or other conditions.

Examples

As an example of how these protocols are used, consider migrating a file. A GDS first selects a file copy to migrate and does an INQUIRE on the copy. This may have already been done as part of another operation (for example, if this is a demand migration, the INQUIRE would have already been done as part of quorum collection).

The GDS does a WRITE_PROPERTIES to mark the copy as migrating, and to indicate that it is to be deleted at commit time. It then uses an INQUIRE to create a copy at the new location. It

does a READ_PROPERTIES on the old copy to get the property list, edits Roe properties if needed, and sends the properties to the new copy with a WRITE_PROPERTIES. It then does a READ on the old copy, specifying the new one as the destination. After this finishes, a SYNC is done on both copies, an UPDATE_ENTRY is done for the entry pointing to the file, and then a COMMIT is done to finalize the changes.

3.6. Discussion

3.6.1. Meeting the Goals of Roe

In section 3.3 we derived a set of goals, based on our thesis statement, that Roe was to meet. Roe's methods for addressing each of these goals are as follows:

- **Network transparency**: Roe supports its own network transparent global directory and provides transparent access to distributed resources through this directory. This ensures network transparency.
- **Simple user model**: Roe appears to users to be a single, globally accessible file system that provides highly available, consistent, sharable files.
- **Consistency**: Atomic transactions are used to ensure internal consistency in any given file or directory copy. The use of weighted voting guarantees that users will always see the latest copy.
- **Enhanced availability**: Roe supports replication of files and directories to increase availability. It also maintains a network model that includes information on host availability and uses this to place resources to maximize availability.
- **Reconfigurability**: Roe uses automatic file placement, file migration, and replication to mask and adapt to failures and other changes in underlying resources. The network model maintains information about the state of these resources and allows placement and migration to be done effectively.
- **Performance**: Roe uses automatic placement and migration to place data near those using it. The network model includes information on host performance and congestion

that can be used to balance load and reduce delays. Directory information is cached to reduce the overhead of name interpretation.

- **Heterogeneity**: Roe supports data conversion between machine boundaries where possible, machine-dependent files, file properties for storing information useful to particular host types, and a modular structure that encourages heterogeneity by minimizing the initial "buy-in" cost.
- **Scalability**: Roe uses algorithms, such as weighted voting, that are able to operate in the presence of partial knowledge. This minimizes the need to maintain network state. Control of access to data is placed with the data. This simplifies the replication and distribution of higher levels, and allows these levels to be spread across the network.

3.6.2. Weaknesses of the Roe Approach

A significant weakness of Roe is its lack of support for autonomy. Once a file is stored in Roe, users have no control over where it will be located. This makes stand-alone operation of a host using Roe unreliable at best. This is not an issue in the LAN environment that we have been concerned with, but it complicates extending Roe to other environments.

This lack of autonomy is the result of three factors: Roe's use of a separate global directory to ensure transparency, replication to increase overall availability, and our insistence on consistency for replicated files and directories.

Two related issues are security and the use of Roe in wide area networks. The current design of Roe doesn't explicitly address security issues, although it would be straightforward to add access control mechanisms given our assumption of a single administrative domain. However, if Roe crosses administrative domains or is on a network that includes untrusted hosts, placement and migration algorithms would need to take security implications and autonomy requirements into account. This is more likely to be an issue in wide area networks. This, combined with autonomy issues in wide area networks, would make it difficult to extend the Roe model to this level. A more appropriate model here might be a separate Roe system controlling resources on each LAN,

with provisions to access Roe systems on other LANs.

Performance may also be an issue in some Roe configurations. Roe includes algorithms to place, migrate, and group data to improve performance. However, the use of a separate distributed global directory, combined with the overhead of ensuring the consistency of file and directory data may result in substantial performance penalties, particularly if Roe is implemented on top of host operating systems, as planned.

Roe caches directory information to reduce the overhead of name lookup. One way to reduce the overhead of ensuring file consistency would be to place advisory locks (which are broken, with notification, on a write) on frequently used files. This would eliminate the need to collect quorums every time these files are opened. The degree of success such a tactic would have would depend on the amount of file sharing, on update rates, and on reference locality. We will examine this issue again in Chapter 7 in the context of data on file usage patterns.

3.6.3. Strengths of the Roe Approach

An important strength of the Roe approach is its support for complete network transparency, and the approach it uses to ensure transparency. This provides a framework that allows Roe to integrate existing solutions for file placement, migration, consistency control, and other problems.

Using these techniques, Roe can transparently integrate new resources, make effective use of existing ones, reconfigure to balance load and adapt to failures, and replicate resources to increase availability. Roe does this while ensuring consistency, network transparent naming, and transparent access, thus supporting a particularly simple user model.

Roe provides extensive support for heterogeneity, including transparent data conversion, machine-dependent files, file properties for extending Roe's file abstraction, and a modular structure that reduces the initial "buy-in" cost. As we will see in Chapter 4, this allows heterogeneous resources to be easily integrated into a network.

Roe is designed to scale well to large LANs. It makes extensive use of algorithms, such as weighted voting, that are able to work in the presence of partial network knowledge. Control of data is placed with the data. This use of distributed control eliminates bottlenecks that occur in centralized designs. The network model and the use of automatic migration can be used to balance load and avoid congested "hot spots" that plague large networks.

3.7. Summary

This chapter has presented a design for Roe, a fully transparent distributed file system for a heterogeneous local area network. Roe appears to users to be a single, globally accessible file system providing highly available, consistent files. Roe uses file replication, atomic transactions, a replicated global directory, a detailed model of the network, automatic placement, and migration to provide full network transparency.

File replication based on weighted voting is used to enhance the availability of Roe files and directories. Weighted voting was chosen because of its simplicity, its support for decentralized control, and its ability to operate in the presence of partial knowledge.

Roe maintains a replicated global directory that is used to name Roe files and directories. The separation of this name space from that of hosts supporting Roe allows Roe to transparently replicate, distribute, and reconfigure files and directories. The global directory is replicated using a modified weighted voting algorithm that supports long-term connections and caching.

A network model encodes information used in placing and migrating files and directories. The model includes information on host congestion, available space, availability, state, and so on. This network model is used to perform automatic file placement and migration. Files are placed based on available space, congestion, topographic, and other considerations. File migration may be used to adjust to changes in underlying network resources, to improve performance by moving data closer to users, and to balance load.

A network is typically made up of a number of dissimilar machines, and Roe recognizes this by providing support for heterogeneity. This includes typed files, conversion between machine boundaries, support for machine-dependent files, file properties to allow Roe's file model to be extended, and a modular structure that eases the integration of new hosts into Roe.

As we have seen in this chapter, the full network transparency supported by the Roe design allows it to integrate these techniques and to use them to provide increased availability, transparent reconfiguration, effective use of resources, a simplified file system model, and performance benefits such as load balancing and migration to reduce overhead. The design provides a strong vindication of our thesis statement.

Chapter 4

The Roe Implementation

4.1. Introduction

An implementation of Roe was undertaken to validate the design presented in Chapter 3. This implementation was also intended to provide an environment for experimenting with other file system designs, with file placement and migration algorithms, and, eventually, a file storage and management facility for the department's network.

It is frequently argued in systems research circles that an implementation is the most effective (and sometimes only) way to validate a system design. It often uncovers problems and complications that were not anticipated in the original design, and points up areas where future work is badly needed. As we will see, the experiences gained in implementing Roe confirm this view.

In this chapter, we first describe the environment in which Roe was implemented. We then sketch out the implementation approach used and present the resultant system. Given this background, we describe problems that were encountered in the implementation and make some suggestions for future systems. Finally, we discuss the extent to which our goals were met and detail lessons learned from the implementation.

4.2. The Implementation Environment

Roe was conceived and designed in the context of a larger research project on a testbed system for distributed file systems. Implementation of this testbed system was started in late 1981. At that time, the University of Rochester Computer Science Department's network consisted of several Data General Eclipses running RIG (a locally developed message passing system), VAXen running BSD UNIX, and a number of Xerox Alto [Thacker 79] workstations running Alto/Mesa. These hosts were connected together with a 3MB Ethernet [Metcalfe 76] (see Figure 4-1). Roe was implemented on all 3 classes of hosts. This gave us an opportunity to explore some of the complications introduced by a heterogeneous environment.

The following section briefly describes the environments provided by RIG, UNIX, and Alto/Mesa. Section 4.2.2 describes the interprocess communication mechanism (IPC) used in the

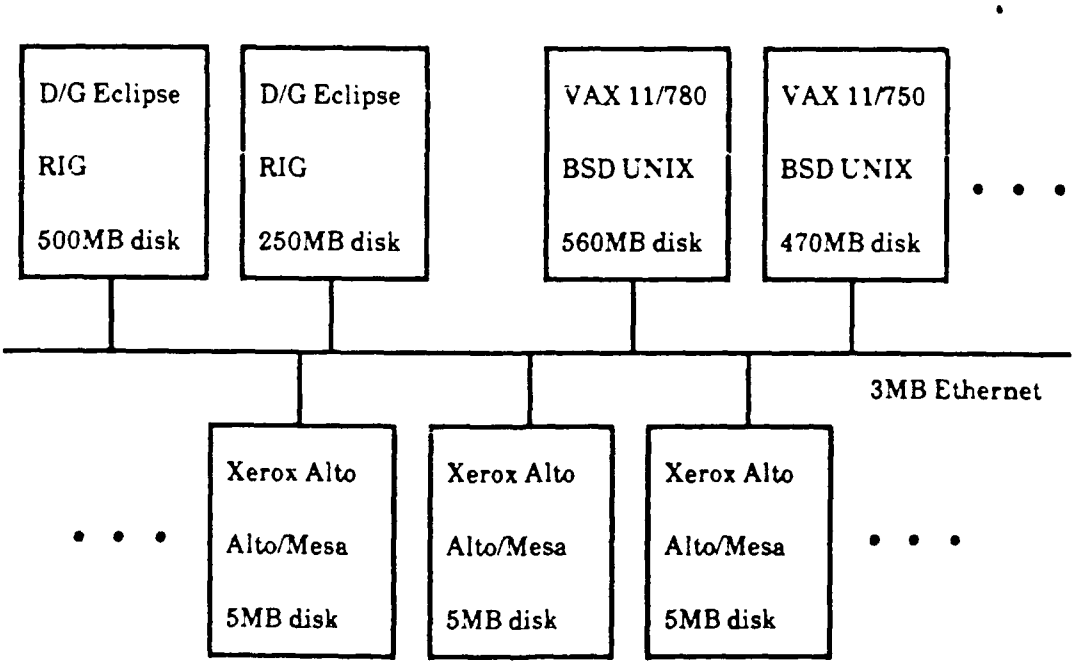


Figure 4-1: U of R Computer Science Department 3MB network (circa 1983)

implementation.

4.2.1. The Host Environments

RIG [Ball 76, Lantz 82] ran on several Data General Eclipse minicomputers. It was a message-based system that provided file service, ARPANET access, printing services and a number of other functions for client hosts (primarily Xerox Altos). RIG supported a tree-structured file system. Files were managed by a file server process and presented to the user as seekable streams of bytes. Requests to read or write a file were limited to 1K bytes per message.

For the developer of RIG software, the primary characteristics of the system were that typed messages were the basic method of communication, process address space was limited (64K bytes), physical memory was limited, context switching was slow, and process creation was very slow. These factors worked together to encourage a structure where one server process was responsible for a resource. Such a server typically handled a number of client sessions simultaneously, multiplexing between them and processing as much as it could of all active sessions before blocking.

The development (and only) language on RIG was bcpl. Bcpl is an ancestor of C, with similar structuring facilities, but with a cleaner syntax and no typing.

UNIX [Ritchie 78] is a general-purpose, interactive time-sharing system. UNIX supports a tree-structured file system containing both files and devices. Files are represented as seekable streams of bytes. Associated with each file is additional information on the size of the file, creation date, protection, and so on.

Process creation in UNIX is generally much cheaper than on RIG, although still significant relative to file access times. The version of UNIX used in this effort (4.1 and 4.2BSD UNIX) supports a large virtual address space, and so allows servers to retain significant amounts of state. C [Kernighan 78] is the language of choice for most system development on UNIX.

The Xerox Alto is a microprogrammable, single-user workstation with a bit-mapped display and mouse. Files on the Alto are seekable streams of items (generally bytes or words) plus a *leader*

page that contains hints about file size, name, file type, and so on. User information may also be stored here.

Development on the Alto was done using Mesa in the Alto/Mesa environment [Mitchell 79, Xerox 79]. The distinguishing features of the Mesa language are strong typing, elaborate facilities for modularization and information hiding, and support for cheap shared-address processes. The language provides monitors to synchronize access to this state. In addition to the programming language, the Alto/Mesa environment provides a number of packages for file access, process management, display manipulation and so on.

4.2.2. Interprocess Communication

When the implementation of Roe was started only RIG, of the 3 environments described above, had a native IPC mechanism that provided a sufficiently rich set of functions. There was, however, an add-on IPC environment developed by Rashid at CMU [Rashid 80] for 4.1BSD UNIX. It was the logical successor to RIG's IPC and was generally compatible with it. This IPC was adapted for our UNIX machines, necessary extensions were made to RIG's IPC and a compatible implementation was done for the Altos. Servers were written for each environment to extend the IPC across the network. This work is described elsewhere [Moore 82].

The remainder of this section describes the facilities provided by the network IPC, presenting only those aspects important to the Roe implementation. For the sake of brevity, we will discuss primarily the UNIX implementation, generally referred to as CMU-IPC. The Alto implementation is basically identical from a user's viewpoint, the differences in the RIG IPC had little effect on the implementation, and the extension across the network was, for the most part, transparent.

CMU-IPC provides two basic types of objects: *ports* and *messages*. A *port* is a FIFO queue of messages. One process may receive messages from a port and any number of processes may send messages to the port (Figure 4-2). All access to a port is through a secure *capability*, which is a process-local name for the port. CMU-IPC notifies processes with capabilities to a port when the port is destroyed or becomes unreachable and the capability becomes invalid.

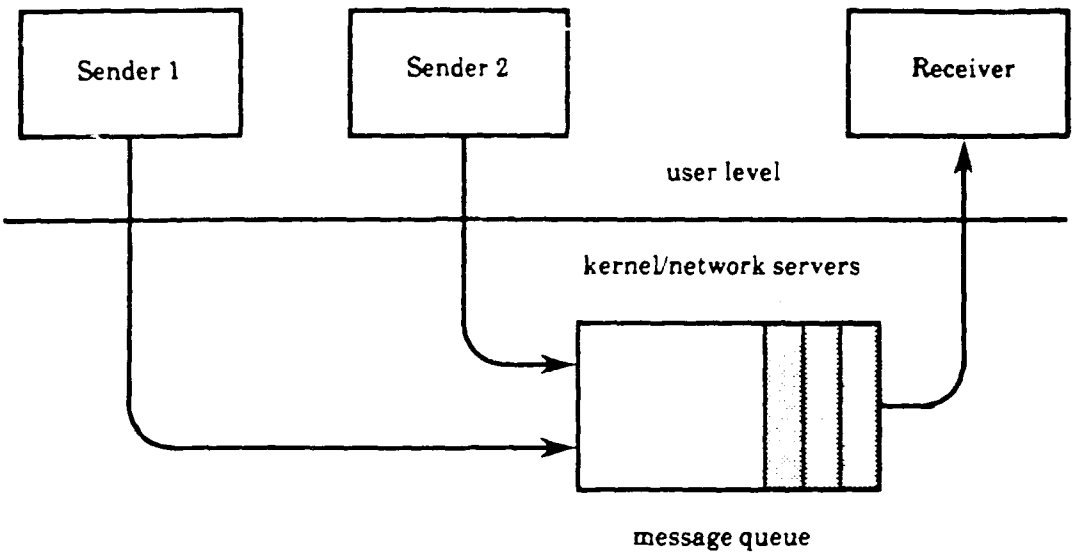


Figure 4-2: A port in CMU-IPC

Ports in CMU-IPC are anonymous. Given the local capability for a port there is no way to determine the location of the process receiving messages sent to the port. This would, in theory, allow the receiving process to change during the lifetime of the port, but the protocol for doing this across the network is non-trivial and was never implemented at Rochester.

A *message* is a collection of typed data objects. Supported data types are currently limited to character, integer, port, uninterpreted, and arrays of any of these. In addition to the data, messages have a priority (normal or emergency), an integer ID (set by the sender), a destination port (where the message will be delivered) and a reply port. Normal messages follow the FIFO queue discipline described above. Emergency messages are also received FIFO, but they are queued ahead of normal messages sent on a port. Messages are delivered reliably.

The queue for a port is limited in size. *Flow control* is invoked when a process attempts to send a normal message to a full port (emergency messages are not flow controlled). One of three flow control actions (specified by the sender) is taken:

- The sender is blocked until there is room for the message; or
- The send fails; or
- The message is accepted and the sender is notified when it is actually queued (limited to 1 outstanding message per sender per port).

It is possible to associate a string name with a port. Other processes may then lookup a port by name and receive send rights to it. If a name lookup request cannot be satisfied locally, the request is broadcast to other machines on the network. There is no other structure in the IPC name space. There are two other ways for a process to gain send rights to a port: the parent of a process may request send rights to the two default ports created with the process (the data and kernel ports) or a process may receive them from another process in a message.

The IPC described above was extended across the net using user-level servers. PUPs [Boggs 80] are used as the underlying transport mechanism. This limits messages to a maximum of 512 bytes on some systems. Typed data in messages is converted to a network standard format (VAX/UNIX) before being placed on the network and converted back when received by a server.

4.3. The Roe Implementation

This section describes the actual implementation of Roe. We first describe the general approach used in the implementation. We then present each of the servers that make up Roe in turn. Finally we summarize what was and was not implemented in the context of the architecture given in Chapter 3.

4.3.1. General Approach

Our general approach was to implement enough of Roe to both validate the approach used and to make the system usable. In practice this means that a fairly complete implementation of the system was done in the UNIX environment and a limited subset of the system was implemented in the RIG and Alto/Mesa environments. This allowed us to validate the overall design, investigate

the heterogeneity aspects of Roe, and implement a working system in a reasonable amount of time.

The structure of the implementation follows closely the architecture given in Chapter 3. Local File Servers and Local Representatives have been implemented for all three host environments. This allows files to be stored on all hosts on the target network. Transaction Coordinators, Local Directory Servers and Global Directory Servers have been implemented in the UNIX environment. Servers are implemented as separate processes, with information exchanged only through messages. This approach has obvious performance disadvantages, but allows for easier development, monitoring, experimentation, and incremental implementation. The two obvious alternatives, implementing Roe as one process per machine and implementing Roe in the kernel of the hosts are less appropriate for a research prototype of this complexity.

We made minimal changes to the host environments. This was in keeping both with our goal of gracefully accommodating heterogeneity and with the limited manpower available. The most striking example of this is our reliance on the file systems of the hosts. Roe files and directories are implemented as files residing on the regular file systems of the hosts. In a similar spirit (and because we were not the sole users), we used CMU-IPC without any modifications as a base for the implementation[1].

Roe has not been released to the general public, but it was implemented with this possibility in mind. There are two important consequences of this: 1) servers never block on user processes or other servers, and 2) a functional implementation was done. We will come back to these points in the following sections.

4.3.2. Roe Server Implementations

[1] As we shall see, this is a classic example of building on sand.

4.3.2.1. Local File Server

Local File Servers were implemented in the UNIX, RIG and Alto/Mesa environment. We will describe first the UNIX implementation and then use this as a base for discussing the other two implementations.

4.3.2.1.1. UNIX

The UNIX Local File Server supports the fileport protocol described in Chapter 3 (with the exception of crash recovery). The UNIX LFS also supports most of the directoryport protocol. This allows users to access files local to UNIX machines (files that lie outside the control of Roe) using the same protocol used to access Roe files.

The initial UNIX LFS created a new process for each user that logged in and for each file that was opened. Dedicating a separate process to each user and file resulted in a very simple implementation. Requests from users could be mapped in a fairly straightforward fashion into UNIX system calls, with no need to worry about a server process having to deal with multiple outstanding requests. Unfortunately, the performance using this approach was not very impressive. The time required to fork a new LFS process (100ms on a VAX 11/750) was an order of magnitude greater than the time required for the new process to open the file (an *fopen* call typically takes 5-10ms under 4.2BSD UNIX) and so dominated the cost of opening a file.

This led us to re-implement the UNIX LFS as a *multiplexed* server. A new process is still forked when a user logs in (for protection purposes), but now all file requests for a given user are handled by a single process. A user may have a number of files open at once, and in various stages of reading and writing. We wished to preserve the appearance of a non-blocking server (that is, an active request on one file shouldn't preclude operations on another file). We do this as follows: When a process receives a request to open a file, it allocates a new port to handle requests for the file and passes the port back to the user. Requests on the file always arrive at this port. Associated with the port is a *processing routine* that is called when a message is received on the port, and a *context* to use in interpreting the message.

```
processor = proc1;
while (TRUE)
{
   receive(message);
   processor(message, context);
}

proc1(message, context)
{
   start processing on the request;
   processor = proc2;
   save context for proc2;
   return;
}

proc2(message, context)
{
   finish processing;
   processor = proc1;
   return;
}
```

Figure 4-3: A Multiplexed Server

All messages are received at a single location and are then dispatched to the processing routine associated with the port. This routine handles as much of the request as possible and, if it isn't able to finish the request without blocking on a message send or receive, sets up a processing routine and context to continue the request. It then returns to the central dispatching loop. The overall structure of a Roe server using this approach is shown in Figure 4-3.

This fairly crude approach allows us to write multiplexed servers without any underlying system or compiler support (an important consideration in a heterogeneous environment). It does, however, result in an implementation that is harder to understand and modify, since the flow of control in the server is much less obvious. We will come back to this in section 4.4.

The LFS maintains a property list with each Roe file (files opened using INQUIRE) that it manages. This list is stored at the front of a file and is read in when the file is first opened. There is no limit on the size of this property list. The structure of a Roe file in UNIX is shown in Figure 4-4. It should be noted that this approach doesn't allow properties to be stored in non-Roe UNIX files. A number of other representations were considered that would have allowed

properties to be associated with any UNIX files, including adding a pointer to a properties page in the UNIX file system inode structure, maintaining a properties "database" containing properties for all files on the system, and maintaining a "properties directory tree" that shadowed the directory tree and files of the associated UNIX file system. These were all eventually dropped because of complexity or performance reasons.

There are actually two types of properties stored in the property list of a file: properties used by the Roe file system and user properties. By convention, Roe properties have names of the form "ROE*." Five of these properties are present in every Roe file and have special meaning to the LFS. "ROEtimestamp," "ROEr_quorum," "ROEw_quorum," and "ROEvotes" are used in the weighted voting algorithm. They must be present in the INQUIRE request when a Roe file is

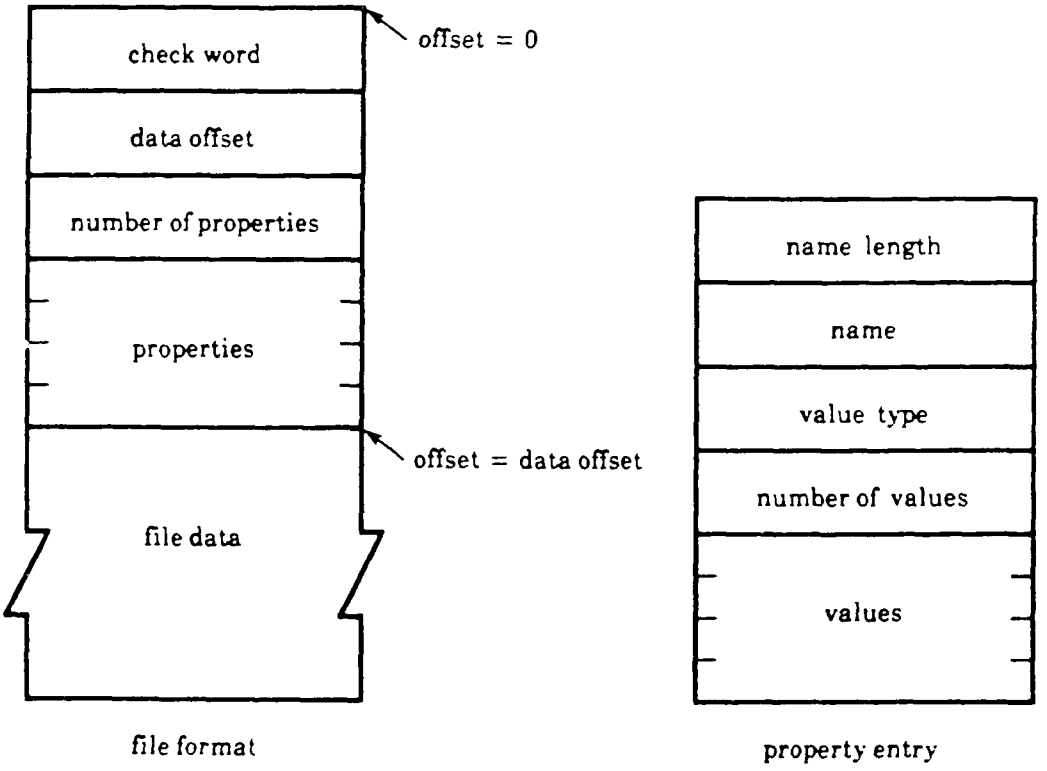


Figure 4-4: Structure of a Roe file on UNIX

created and are returned by the LFS when a Roe file is opened. If the file is opened for write access, the LFS increments the timestamp ("ROEtimestamp"). "ROEtype" specifies the type of the data in the file. It is used to determine the size of the basic object in the file and to type messages containing data.

The UNIX LFS supports atomic actions on files. If a file is opened with atomic access, a copy of the file will be made the first time it is written and all work will be done on this copy[2]. At commit time the copy replaces the original. The *fsync* system call is used to ensure that a consistent copy of the file is on disk. Full crash recovery is, however, not currently implemented.

4.3.2.1.2. RIG

The RIG Local File Server was also heavily multiplexed, with one process handling all requests for a given machine. On RIG, file system requests were done using messages to the file server process. This resulted in more fragmentation in LFS processing routines, with the corresponding complexity in development and maintenance.

In parallel with the development of the RIG LFS, plans were made to rewrite the RIG file server. This rewrite would have provided a more robust on-disk structure, support for atomic transactions and stable storage, support for properties, and information on disk space available. When it became clear that RIG was to be phased out, these plans were dropped, along with the remainder of the Roe implementation effort on RIG. Of the proposed file system changes, only support for atomicity would have been strictly necessary for Roe.

4.3.2.1.3. Alto/Mesa

In the Alto/Mesa environment we were able to take advantage of the lightweight processes provided by Mesa. Lightweight processes allowed us to avoid explicit multiplexing. The resultant implementation is a comparative joy to read. The Alto/Mesa LFS implements the entire LFS

[2]This is not a particularly clever implementation, but it has the significant advantage of requiring no changes to the UNIX file system to implement.

protocol (*including* crash recovery) and took considerably less time to implement than either the RIG or UNIX LFSs. Each user login session and each opened file is handled by a separate process. In situations where it is necessary to access shared state (the file lock table and the intentions logs), Mesa monitors are used for synchronization. Properties are stored in the Alto leader page.

4.3.2.2. Local Representative

The UNIX Local Representative supports two requests: 1) SPAWN a new Transaction Coordinator and 2) return STATUS information on the local pack and machine. The TC is part of the LR in the UNIX implementation and so SPAWN reduces to a *fork* call. Returning status information is somewhat more complicated.

When the LR starts up, it reads a configuration file containing the name of the pack it is representing, along with information on the operating system type, machine type, availability, and basic page access time. This is the status information that doesn't change while the LR is running. When a STATUS request is received, the LR gets the amount of disk space currently available (using the shell-level command "df") and estimates the current page access time (we use as a rough estimate (basic page access time)*(load average)). This information is returned along with the static status information.

The RIG and Alto LRs support a reduced form of the status command. On the Alto only the available space changes dynamically and on RIG all returned status information is static (and hence not very accurate).

4.3.2.3. Transaction Coordinator

The Roe Transaction Coordinator takes a user request on an open file suite and distributes the request to the appropriate file copies in the suite. A TC has been implemented for UNIX. It has a multiplexed structure similar to the LFS, but with the added complication that there is an error handling routine associated with each allocated port. As with the processing routine, the error handling routine may be changed as needed. This allows us to easily express the different error

handling requirements of, for example, a READ versus a COMMIT. There is at most one TC for each active user of Roe, and so a given TC may be handling multiple file suites simultaneously. The multiplexed structure keeps the TC from becoming a bottleneck.

Reads on a file suite are sent to one copy of the file. Writes go to all accessible copies. If a copy of the file fails, it is removed from the quorum. If a quorum no longer exists for an operation, then the operation fails. If the copy being read fails in the middle of a read operation, an attempt is made to switch to another copy. This change is transparent to the user process. The TC maintains for each file suite the current file offset, calculated based on the last SEEK and on accumulated READ and WRITE operations. Data read from a file are sent first to the TC, the offset is updated, and then the data are forwarded to the user. If a copy fails in mid-stream, another copy is selected, a SEEK is done and a READ is started from the point where the last READ failed. The TC's internal file offset is also used to do a SEEK before doing a WRITE (to all copies) that follows a READ (to a single copy).

4.3.2.4. Local Directory Server

The UNIX Local Directory Server manages, for Global Directory Servers, Roe directories on a host. It is implemented as a single process per host. Roe directories are stored as regular UNIX files, with one directory node copy per file. A directory copy contains some administrative information, a property list, and the directory entries. The structure of a directory file is shown in Figure 4-5. Figure 4-5 also shows a sample directory entry (this is the on-disk format, with each of the strings being stored in <length, characters> format). This entry describes a plain file. Other entry types describe directories, user entries and machine-dependent files. These types are described in more detail in Chapter 3.

When a directory node is opened, the directory is read into memory and cached in the LDS. Since directories are typically very small and we expect references to be fairly localized (Chapter 6), the memory and processing cost involved here are insignificant. Reads are handled from memory. Updates are made first to the in-memory copy of the node and then the modified node is

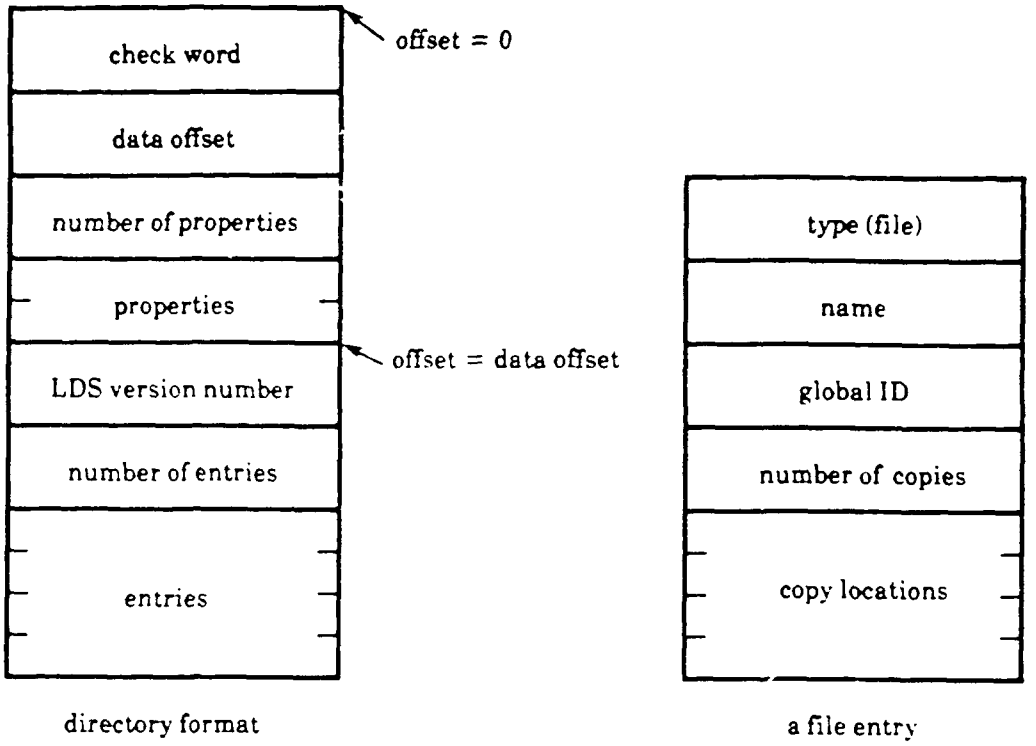


Figure 4-5: Structure of a Roe directory

forced to disk. Atomic updates are supported using the protocol described in Chapter 3. There is currently a limit of one such update pending per directory node, although it may contain multiple operations. Crash recovery has not been implemented.

When a directory node copy is first opened, the LDS allocates a new port that will be associated with the node and returns it to the opener. Requests to open an already opened node are handled by simply making a note of the GDS making the request and then returning the port already associated with the node. The LDS maintains a list of all processes with access to a node. This list is used to notify GDSs when a copy is out of date or otherwise potentially invalid. This list is also used to keep track of the number of GDSs actually connected to a node. When this number drops to zero (because of CLOSEs or lost connections), the node is closed.

The LDS also associates with each node a *state*. This state may take one of three values:

- CURRENT: The node copy has been read or updated and has not lost connections to GDSs.
- STALE: The LDS has been unable to make at least one update to the node (due to disk problems, for example).
- UNKNOWN: Before a node has seen any reads or updates or after it has lost connections to GDSs.

If a node is STALE, requests to read or write the node are rejected until the copy is updated by a GDS. The other two states (CURRENT and UNKNOWN) are sent to the GDS along with weighted voting information when a node is opened (see Chapter 3).

4.3.2.5. Global Directory Server

The Global Directory Server is the glue that holds Roe together. It maps user requests into *requests on* Roe file and directory suites, maintains the network model, determines initial file placement and manages migration. The GDS described here is a concrete realization of the GDS described in Chapter 3. Its structures and algorithms generally follow closely those presented in Chapter 3. One exception is the representation of information on host storage capabilities. The implementation of Roe described here organizes storage by *pack*, rather than by host. A *pack* is a unit of storage whose contents and space may be treated at a whole by Roe. A pack may or may not actually be tied to a particular host. It can be thought of as corresponding, on Altos, to a physical disk that may be moved from machine to machine or, on UNIX systems, to a file system.

In the next section we describe the initialization procedures for the UNIX implementation of the GDS. Following sections cover the network model maintained by the GDS, file and directory suite management (opening files, deleting them, updating obsolete copies and so on), and file placement and migration.

4.3.2.5.1. Initialization

When Roe is first started up on a UNIX host, a Local File Server, a Local Representative and, optionally, a Local Directory Server and Global Directory Server are created. When a GDS is started up, it first reads a *GDS state file* containing the last GDS *incarnation number* on this machine and a record describing the location of the root of the directory tree. The incarnation number is incremented each time a GDS is forked and is used when generating unique file identifiers. The record describing the root contains the file ID and packs of the copies of the root (in the format used by the LDS to store directory entries). The GDS uses this information to open the root of the directory tree.

The GDS reads two other files on startup. The first, the *local pack file* contains the name that the local pack will be known under. This name is unique across Roe. The GDS uses this name when generating unique file IDs. It is also used as the default location for newly created files, directories and processes. The *network status file* contains, for the network, information on packs and their hosts. Each entry contains the name of a pack, the machine type and operating system of its host, the pack availability, and access time and free space estimates. This information is used to initialize the network model maintained by the GDS. It is updated when packs are contacted.

After the GDS on a host has finished its initialization, it asserts the CMU-IPC name "pack.roe." A user wishing to contact Roe does a name lookup on "pack.roe" and logs in. If there is a GDS server running on the user's local host, it will be used. Otherwise a broadcast name lookup is done. The GDS on the first host to respond will be used.

When a user logs in, a new GDS process is forked for the user. This GDS process handles all user requests that don't involve internal file operations (these are handled by the TC). The GDS is not multiplexed, but instead handles user requests serially. This is generally not a problem, since the GDS, unlike the LFS and TC, isn't involved in ongoing transactions.

4.3.2.5.2. The Network Model

When the GDS receives a user request to, for example, create a file, it contacts LDS and LFS processes to read and create directory entries and to create file copies. Each GDS maintains a *network model* containing information about the state of the hosts and servers in the network. This state is used to contact servers and to place newly created objects.

The network model is organized by pack. The following information is kept for each active pack:

- Pack name.
- Host status (up, down or unknown).
- Last update (time of last STATUS request to the LR for this pack).
- LDS, LR and LFS ports for this pack (if available).
- Machine type (VAX, Alto, ...).
- Host operating system (UNIX, RIG, ...).
- Host availability.
- Free space (dynamic estimate).
- Delay (a dynamic estimate of the time for a host to complete a page read).

When a pack record is first read in from the network status file, it is given a host status of "unknown." The first time the GDS acquires a port for one of the pack's local servers, the status is changed to "up." If connections to all local servers on a pack are broken, the host is marked as being "down" (or at least inaccessible). A GDS will periodically try to recontact down hosts. The machine type, host operating system and availability information are from the network status file, or, if this is a machine that wasn't in the status file, from the pack's LR. The free space and delay information are updated periodically using the replies to STATUS requests sent to up LRs.

GDS processes maintain current state information for packs that they expect to be using in the near future. Packs are ordered on an LRU basis. Connections to a new pack are added when a request is made to access or place an object on the pack. A GDS process periodically polls LRs on active packs to update free space and delay estimates. This is currently done every 10 minutes.

Polling for network status can be expensive, particularly in a large network. The GDS tries to control this overhead by only contacting a limited number of recently used packs. A better solution would be to have a GDS process maintain connections with a limited number of LRs and to have these LRs send notifications when there are significant changes in free space or delay estimates. This has the advantage of both decreasing the amount of unnecessary network traffic and of providing timely notification of important changes. This approach was not used because it would have required potentially major structural changes in the non-multiplexed GDS.

The general approach used in this implementation of the network model was one of "minimal impact" and attention to scaling issues. A GDS process only acquires connections to packs as they are needed. The obvious alternative, attempting to acquire connections to all known packs at startup time, would provide the GDS with considerably more information about the network, but also incurs substantial costs from name lookup and connection establishment. These costs become prohibitive in large networks. A GDS also only maintains current state for recently accessed packs. We make no attempt to maintain enough information to do optimal file and directory placement. However, the approach used, when combined with the placement heuristics that will be described in the following subsections, can be expected to provide effective placement in our environment with relatively little overhead.

4.3.2.5.3. File and Directory Management

The major task of the GDS is to open files for the user. When a request is received to open an existing file, the GDS first reads the directory entry for the file. This is done using connections maintained by the GDS to copies of the user's current working directory and to other recently used directories. If this is a regular file, the GDS then sends an INQUIRE request to all copies of the file and collects the responses. If a quorum is received, the result can be sent to the users's Transaction Coordinator. Before this is done, the GDS checks to make sure that all copies of the files are current. Obsolete copies are updated by rewriting both the file data and properties. Only the voting information is left unchanged. Once the TC receives the ports and other information for the file quorum, it can accept user requests to access the file data. The sequence described above

is shown, for an unreplicated file, in Figure 3-4.

Opening files is not always as simple as the description given above suggests. If the file doesn't exist and is being opened with create access, the GDS first composes a unique file ID for the new file. The file ID has the form

<local_pack_name>.<incarnation_number>.<sequence_number>.

Here the incarnation number is from the GDS state file and the sequence number is incremented for each new file created by this GDS incarnation. The GDS then selects locations for copies of the new file (see the next subsection), places the copies and creates a directory entry describing the file.

The file being opened may be a *machine-dependent* file (returns data that depend on the machine and operating system being used). If the file is an existing file, the machine-dependent characteristic will be specified in the directory entry for the file. The entry will also contain a list of file suites, one suite for each machine and operating system combination that the file supports. The GDS attempts to match the target machine and operating system with one of the file suites in this list. The target machine type and operating system may be specified as properties in the OPEN request. If they are not given, the GDS uses the machine type and operating system of the user's *home pack*. This is the pack that the user supplies when he logs in. If no match exists, the open fails. A machine-dependent file may be created by specifying a machine type and operating system when a file is created.

Another complication is file migration. The open request may trigger migration of this file. This is done before the file is opened. Finally, there may be no Transaction Coordinator active for this user when the file is opened. In this case, the GDS sends a SPAWN request to the Local Representative on the user's home pack to create one.

A GDS process maintains connections to directories it expects to be using again soon. This is presently limited to the current working directory and its ancestors, but could easily be expanded.

When a directory node is first referenced, the GDS reads from its parent the entry describing the node. It then contacts the copies of the node and collects a quorum. If any of the copies of the directory are obsolete, they are updated at this time. The GDS selects a copy of the node (generally the first one to respond to the open request) and uses it for subsequent reads. Updates are applied to all copies in the quorum. Emergency messages are used to notify a GDS when a copy fails. Failed copies are removed from the quorum. If a quorum no longer exists, the node is closed.

4.3.2.5.4. Initial Placement and Migration

When creating a file or directory, the user may specify the number of copies to create, locations to place the copies, and the voting configuration to use. This information is a property of the object and is given in the property list that is part of the OPEN or CREATE_DIR request. If the user omits this information, defaults will be used. In this case, the GDS must select locations for copies. The network model plays an important part in this selection process.

The GDS attempts to place files and directories to minimize access time, subject to loose availability constraints. The first copy of a new object goes on the user's home pack. We are assuming here that access to local resources takes less time than access across the network. This is not true in general, but is definitely the case in our environment. Locations for the rest of the copies are found by picking from the active packs represented in the network model those with the lowest delay, subject to static minimum free space and availability constraints. C-like pseudo-code for the placement algorithm is given in Figure 4-6.

The GDS supports migration on reference for files. If a file is opened by a user and a copy does not exist on his home pack, a copy will be moved there. Migration proceeds as follows:

(1) Open the copy to be moved, specifying DELETING and MIGRATING properties (this causes future INQUIRIES on the copy to be rejected and marks the copy for delete on close).

```
pack[0] = user's home pack;
for (i = 1; i < number of copies; i++)
{
   pack[i] = a pack such that it has not yet been selected
        && delay < all other unselected packs
        && free space > space threshold
        && availability > availability threshold
        && pack state != down;
}
```

Figure 4-6: Initial placement of file and directory copies

(2) Tentatively create a copy on the user's home pack, copying over all data and properties.

(3) Tentatively update the directory entry for the file.

(4) Atomically close the old file copy (causing it to be deleted) and commit the directory changes and the new copy.

4.3.3. What the Implementation Provides

The previous sections of Chapter 4 have covered the Roe implementation at a fairly low level. This section presents a high-level view of the capabilities of the implementation.

The Roe implementation provides:

- Fully network transparent access to files and directories. The user has no need to know any of the details of the underlying network, or even if there is one.
- File and directory replication. Both files and directories may be replicated and distributed across the network. This can be done under user control or automatically by Roe.
- Automatic placement of files and directory copies based on user location, access delay, free space and availability considerations.
- Transparent migration of files. The implementation provides migration on reference, with a copy of a file being moved to the user's home pack when the file is opened.
- Transparent reconfiguration to adjust for failed, recovering, and new hosts. Host failures during file opens, reads and writes are masked (if possible). Hosts are recontacted when

they recover. New hosts may be added to the network with no user-visible changes.

- Limited support for atomicity. Atomic operations on individual files are supported. However, only the Alto implementation provides crash recovery.
- File properties for UNIX and Alto files, for use by both the user and the Roe file system.
- Support for heterogeneity. UNIX, RIG and Alto hosts all provide storage for and access to Roe files. UNIX hosts implement the Roe directory. File and property data are typed. In most cases, these data are transformed from one host representation to another when it crosses machine boundaries, and so the heterogeneous nature of the network is masked. For cases where this masking is not possible or desirable, the implementation provides machine-dependent files that return data that depend on the machine and operating system. Roe can automatically select, based on the location of the user, the correct version of a machine-dependent file.

4.3.4. Implementation Weaknesses and Omissions

The Roe implementation is a powerful validation of the architecture described in Chapter 3, but it is by no means complete. Previous experience has shown that developing a system of this size to the point where it can be released takes far more effort than one or two people can provide. For example, the LOCUS distributed operating system is reported to have taken in excess of 70 man-years to implement. Roe has about 4 man years of effort invested in it, and so omissions and weaknesses are to be expected. The major areas where work remains to be done on Roe are as follows:

- Error and crash recovery. Roe supports protocols for atomic transactions on replicated objects but, with the exception of the Alto LFS, doesn't support crash recovery. Error handling, particularly in the area of failure during transaction commit, needs work. These are problems that have received a considerable amount of attention in the literature, though (see Chapter 2). There are a number of solutions available that would be appropriate in the Roe environment. A lack of time and resources, combined with the

presence of other, more pressing problems, prevented us from fully implementing any of them.

- Protocol omissions. Some parts of the Roe protocols described in Chapter 3 were omitted, again due to lack of time. The omissions are: ENUMERATE (get a list of file or directory copies on a host), CREATE_ENTRY (to create a user entry), and NOTIFY (support for directory copy caching and joining).
- Security and protection. There is no way in the current implementation for one Roe user to protect files from another user. Further, file and directory properties used by Roe are not distinguished from user properties and so are not protected from users. The latter problem can be easily addressed in the Transaction Coordinator, but adding higher level protection for Roe files and directories would require network authentication facilities, support for file ownership and access rights specifications (kept, for example, in the property list of the file), and the addition of host security information to the network model.
- Initial file and directory placement. The current placement algorithm uses static availability and space thresholds. Better results would be obtained if these were adjusted depending on the state of the network. Along similar lines, there is currently no way for a user to specify the relative importance of performance, availability and storage costs for a file.
- File and directory migration. The current implementation does not support directory migration at all. This is relatively straightforward, except for the cases of migrating in-use directories (connected GDSs must be notified of the move) and migrating a copy of the root (connected GDSs should be notified so that they can update their state files). File migration is limited to the migration-on-demand algorithm described above. More sophisticated algorithms would require the LFS to keep usage information (as part of a file's property list), support for iterating over files by pack, and the implementation of "forwarding addresses."

- Interface libraries. Roe uses a uniform protocol across the network that is not, in general, compatible with a host operating system's file access protocol. Writing an interface library that converts between these protocols would make Roe much more accessible from a host. This has not yet been done for any of the systems on which Roe has been implemented, although, as we described in Chapter 3, the flexibility of the Roe protocol and support for properties should make this straightforward.
- GDS path interpretation. The GDS isn't able to handle paths with multiple directory specifications.
- GDS directory state. The GDS maintains connections to directories from the root of the directory tree to the current working directory. Maintaining a tree of recently accessed directories and their ancestors, combined with a path interpretation algorithm that short-circuited evaluation of path components that were already cached, could be expected to significantly decrease open overheads under some conditions.
- Network reconfiguration. The procedure to add a new pack to the network is somewhat awkward. The network status file for each GDS is updated to include the new pack and then the initial GDSs on hosts are killed and restarted (this doesn't affect transactions in progress). A new pack will also be used by a GDS after an object is explicitly placed on the pack. An alternative to all of this would be to have the network status file shared by all GDSs (as a Roe file) and to have a mechanism to inform active GDSs of significant changes.
- Network state maintenance. Dynamic network state (space used and delay information) is collected by GDSs using synchronous polling of Local Representatives. This is undesirable from a message traffic standpoint and may also lead to delays in processing user requests. Alternatives are discussed in section 4.3.2.5.2.
- Network structure. GDSs have no knowledge of the actual structure and implementation of the underlying network. This is fine in the current implementation, since remote hosts on an Ethernet are all equally easy to access, but is not acceptable in general.

- Directory updates. The current LDS implementation only allows one transaction to be outstanding on a directory at a time. Most directories are updated slowly and so this is not important, but there are significant exceptions (see Chapter 6 for an example).
- Obsolete directory copies. Obsolete directory copies are updated when the directory node is first opened, but there is no mechanism in the current implementation to add a copy that becomes available after the node is opened.
- Performance. The current implementations of both Roe and CMU-IPC are, in many senses, prototypes. With the exception of multiplexed servers, no attempt has been made to optimize performance. The results are predictable. Section 4.5 presents some figures on the performance of Roe and makes some suggestions for improvements.

4.4. Implementation Difficulties

There have been relatively few large scale distributed system implementations, particularly in heterogeneous environments. One consequence of this is that there is no good understanding of techniques appropriate for implementing such systems, or of the problems most likely to arise. These are areas of current research [Notkin 87]. The distribution strategies and algorithms used by Roe are all straightforward, and were picked, in part, for their orthogonality and ease of implementation. Despite this, implementing Roe was non-trivial. It consumed far more time and effort than we had expected. Part of this was due to the heterogeneous nature of our environment, and to the lack of a common language and common system interfaces for low-level servers. However, a larger part can be attributed to a lack of appropriate implementation tools, and to problems with existing tools, approaches, and environments.

In the following sections we will describe the problems and experiences we had in actually implementing Roe once the system was designed. The emphasis here is on problems one would encounter in implementing *any* distributed application in this environment. Our observations on Roe-related experiences will be presented in section 4.6.

The two major areas of difficulty were with implementing and maintaining multiplexed servers and with CMU-IPC. We discuss each of these in the following two sections and then present a solution that deals with both problems. We then discuss changes to the IPC that would make future implementation easier. Finally, we briefly describe other implementation problems that arose.

4.4.1. Multiplexed Servers

Most of Roe was implemented using *multiplexed servers*. In a multiplexed server, one process handles multiple requests in various stages of completion, switching between them to avoid blocking on one request when it is possible to make progress on others. This approach is described in section 4.3.2.1.1 and shown in Figure 4-3. Using multiplexed servers increases the concurrency allowed in Roe without incurring the high process creation overheads imposed by many operating systems. Unfortunately, the multiplexing approach requires that any particular processing routine be non-blocking and so processing for many requests must be broken up into a number of explicit steps. As Figure 4-3 shows, this fragmentation, along with the need to explicitly store and restore state and to manage storage across multiple routines, complicates the structure of the server. This makes servers more difficult to implement, understand, and modify.

Further complications were introduced by our use of a buffered message passing system with finite length queues and asynchronous error notification. Finite length queues are generally of no help when implementing non-blocking servers. In some situations we chose to avoid using multiplexing to deal with finite length queues by queuing data internally when a port filled up and then sending the data later. Using internal queues simplified the structure of servers, but it made it more difficult to handle errors and exceptions (particularly CANCEL for stream requests) and complicated management of queued ports (see the next section).

There are alternatives to multiplexed servers. One is to provide compiler support for "lightweight" processes (sometimes referred to as *threads* or *tasks*). These are shared-state processes that may be created and destroyed with relatively little overhead. Lightweight processes are provided by the Alto/Mesa compiler and environment. Lynx [Scott 85] is another example of a compiler that

supports lightweight processes. In the case of Lynx, this facility was prompted by the difficulty of implementing multiplexed servers. We chose not to use compiler support on RIG and UNIX because it wasn't available, we needed a solution that required minimal effort to implement and, because of the heterogeneous nature of our environment, needed a solution that would work with several languages.

We did, however, use lightweight processes when we implemented the Alto/Mesa Local File Server. This allows us to compare the two approaches. While it is impossible to give a direct comparison of sizes because of differing library support, the Alto LFS "feels" smaller and simpler than either the UNIX or RIG LFSs, despite the Alto LFS being a more complete implementation. It took less time to implement (roughly half of what it took to implement either the UNIX or RIG LFS), is easier to read, is far easier to maintain, and allows more concurrency than the UNIX implementation (particularly for stream requests).

Another alternative to multiplexed servers is to provide a library that allows lightweight processes to be constructed explicitly. One example of this is the tasking package implemented by Stroustrup for C++ [Stroustrup 84]. A similar package is now available and being used at Rochester [Mayer 86]. While we did not implement any Roe servers using this package, we did experiment with using it in conjunction with CMU-IPC. We will come back to this after we describe our experiences with CMU-IPC.

4.4.2. IPC Problems

A brief description of CMU-IPC was given in section 4.2.2. In this section we evaluate CMU-IPC in light of our experiences in implementing Roe. It should be noted that CMU-IPC is closely related to the Accent [Rashid 81] and Mach [Young 87] IPC facilities. Hence, many of the observations presented below will apply to these two environments.

There were some aspects of CMU-IPC that we found quite useful in implementing Roe. The ability to send typed data and have the data transformed appropriately at machine boundaries allowed us to ignore the heterogeneous nature of the network, even at the local server level, when it was

not important. Strongly typed messages also provide a substantial amount of implicit error checking.

Notification of dead or unreachable servers was assumed in the design of Roe and used heavily in the implementation. It allows Roe servers to keep a sufficiently current network model containing the status of an appropriate subset of hosts and servers. If this facility had not been present, we would have implemented something like it.

Roe makes extensive use of the asynchronous message passing features of CMU-IPC. Statically, most of the Roe protocol could be handled, with no loss of functionality, using a simple RPC (remote procedure call) mechanism [Birrell 84]. However, two of the exceptions, collecting quorums and streaming data, are also two of the most heavily used interfaces in Roe and so they determine the overall performance of the system. Quorums in Roe are collected asynchronously, with requests being sent out to all copies before responses are collected. This allows processing to proceed concurrently at the copies and so decreases open overhead. Using an RPC mechanism here would result in the loss of this concurrency. The streaming WRITE protocol allows us to transfer large amounts of data without the need for multiple high-level requests or acknowledgments. Our experience, then, is that asynchronous message passing has its benefits. It placed no constraints on the structure of our system, and we were able to use this flexibility to our advantage. RPC, despite its attractive simplicity, is not always the best paradigm.

Not all of our experiences with CMU-IPC were as positive as the ones described above. Overall, we found CMU-IPC to be the weakest of the tools used in the implementation. Some of the problems were weaknesses of the particular implementation we used, some can be attributed to the way that we used the IPC, and others to basic design flaws. The problems we had fell into four broad areas: port allocation and deallocation, interpreting emergency messages, sending messages, and name lookup. We discuss each of these areas below.

Roe servers allocate a new port to handle requests for each object opened. The port is deallocated when the object is closed. This allows Roe servers to handle an arbitrary number of objects (up to

the port allocation limit imposed by CMU-IPC) without having to pre-allocate resources and provides a convenient way to associate messages with objects. Unfortunately, this approach interacts badly with the port allocation and mapping algorithms used by CMU-IPC and with emergency message-based link failure notification.

When CMU-IPC delivers a message to a process, it maps the ports in the message to process local capabilities. These capabilities are small integers and are reused after a process deallocates the port associated with the capability or, for a port owned by another process, after the process discards send rights to or loses contact with a port. It is possible that a server may have messages that are being sent or are queued to be sent that contain this port. If the local capability is reused before the messages are actually sent, these messages will be delivered with the incorrect port information (or perhaps to the wrong port). This sort of race condition is generally not reproducible and so is very difficult to track down. The lack of association between local capability names and port locations and owners enforced by CMU-IPC complicates the process. This is a situation where the explicit <process number><port number> specification used by RIG would have made implementation easier.

It is not just Roe that has difficulty with mapping port capabilities. What happens if a message containing a port has been accepted by CMU-IPC and is in transit when the port is deallocated and then reallocated (perhaps by another process)? In this case, the message may contain a reference to a port that the sending process did not have access to, thereby circumventing the CMU-IPC protection mechanisms[3].

It is, of course, possible to keep this from happening, at least inside CMU-IPC, by explicitly checking all in-transit messages when a port is deallocated and invalidating references to it. This is done for messages queued on the port's home machine. One side effect of this check is that, on UNIX machines, deallocating a port takes 4-10 times what it takes to send and receive a local message. The check could be extended across the network but, given the user code

[3]At one point, Roe, intermachine mail and news were all using CMU-IPC. There were several instances of personal letters and news postings being inserted into newly created Roe files.

implementation of the network servers, this would significantly increase the already high cost of port deallocation. The point here is not that it can't be done but rather that CMU-IPC's use of a small name space for ports with reusable names makes it difficult to do correctly, both inside CMU-IPC and in processes that use it.

As mentioned earlier, Roe servers deallocate a port associated with an object when the object is closed. This introduces another interesting race condition. If the Roe server sends a SUCCESS response to the CLOSE request and then deallocates the port associated with the object, the intended recipient of the success response will receive an emergency message notifying it of the death of the port. Depending on the sequence of events and the particular IPC implementation, the SUCCESS message will either be received before the emergency message, after it (but with no indication as to the source), or not at all. The problem here is that the server would like to deallocate the port so that it can use it for future requests, but has no way of knowing if there is still interest in the port. Since there is no easy way, given the current Roe implementation, to reliably deliver responses to CLOSE requests, we have dropped the response from the protocol. Unfortunately, this still leaves other race conditions. For example, the sender of the CLOSE may not get a chance to deallocate the port before the server receives the request and does its deallocation. If the sender then reuses this capability for another purpose it may get a spurious emergency message concerning the capability. However, this is rarer and generally easier to deal with than the CLOSE response race condition.

The problems described above are not due to any particular failing of CMU-IPC. Rather, they are caused by the interaction of dynamic port allocation, reusable capabilities, and the inability of a process to determine outside interest in one of its ports.

CMU-IPC emergency messages, particularly emergency messages that contain notification of link failures, can be difficult to interpret. As we saw above, it isn't possible to tell *why* an emergency message is reporting a link failure. Was the port deallocated, did the process die, or was the host unreachable? Further, since emergency messages are delivered before any queued regular messages, it isn't possible to tell *when* the emergency message was sent relative to the other data in

the stream. Should we wait for more, or have we received everything outstanding on this port? Finally, it is difficult to control *who* receives link failure notifications. These emergency messages come in on a special capability (since the capability they refer to no longer exists). This makes it awkward to integrate multiple packages that use CMU-IPC. An example would be integrating the Transaction Coordinator with a user process. The TC uses notification of link failures to maintain quorums, but if it is part of another application it can't be assured of getting them.

Sending a message in CMU-IPC, as in many IPCs, can be a complicated affair, particularly when compared to a procedure call. The major complication is the relatively large number of possible failure modes. The initial send may fail (perhaps because of an invalid port or unreachable destination). The destination process may die or become unreachable before the message is received, or before a reply can be sent. The destination process may die after a reply has been sent but before we receive the result. Finally, the result may itself be a failure notification. Each of these possible errors must be guarded against. This is one area where an RPC paradigm would simplify implementation.

Our final area of difficulty was with the unstructured string name space maintained by our network IPC implementation. A process may associate any name with a port, provided the name is not already being used on that host. In the case of Roe local servers, the name asserted is a combination of the server type and the host name. For example, an LDS started for pack "coho" asserts the name "lds.coho." Suppose a GDS wishes to contact Coho's local directory server. It does a name lookup on "lds.coho." If the name is not found locally (the GDS is not on Coho), a broadcast is done to find the name. There is no way to specify that a name be looked up on a particular host, so what we have done is to encode the needed information in the server name. The broadcast, which is processed by every name server on the network, is excessive for our purposes.

Consider now the case of a user trying to locate a GDS. Each copy of the replicated Roe GDS asserts the same name, "pack.roe." When a user does a name lookup, the GDS on the first host to respond is used. In the case where the GDS is local, no broadcast is needed. In any case, the user doesn't need special knowledge of the network. The name lookup facility is appropriate for

this situation. However, for some purposes contacting just one GDS isn't enough. It would occasionally be useful in Roe to notify a number of active GDSs of a change in the network (for example, the addition of a host). The name lookup facility would appear to be a natural way to do this, but it isn't generally possible to do this.

4.4.3. A Solution

In this section we present a proposed solution to some of the problems described in the last two sections. Our central goals in designing this solution were avoiding the complexity of multiplexed servers and providing simpler communication primitives than the ones supplied by CMU-IPC, while still maintaining its asynchronous message passing features. We were constrained by our unwillingness to make modifications to CMU-IPC (it was being used by others and would be difficult to modify in any case) and by our need for a language independent solution.

Central to our solution is a library that provides support for lightweight processes ("tasks") inside a conventional process. This approach to writing servers is not new. The novelty here lies in our use of the tasking package to substantially simplify the user's view of CMU-IPC. This will be described below. Our solution was inspired by the C++ tasking class [Stroustrup 84] and by the observation that a Roe server can be organized as a collection of independent tasks, each of which is dedicated to a single object. Allowing a dedicated task to block on message sends and receives will not affect service for other users and objects. With this approach, new tasks are created for each new object that the server is asked to managed (for example, an LFS would create a new task for each successful LOGIN and OPEN). This is done explicitly by the task that processes the request to create a new object. Tasks are destroyed when the object is closed.

With this solution, a server has a central dispatcher that receives all messages sent to the server. To receive a message, an active task calls a special version of receive that puts the task at the head of the *receive list* (the list of tasks waiting for a message) and then returns control to the central dispatcher. The dispatcher does the actual message receive. When a normal message is received on a port, the dispatcher removes from the receive list the task that last did a receive on

the port and returns control to the task, passing it the message. This task runs until it blocks on another receive, a message send, or until it exits.

When a task does a message send, it can specify a blocking or a non-blocking send. If the destination port is full and the task is willing to block (the normal case in a Roe server), the task is placed at the end of the *send list* and control is returned to the central dispatcher. When CMU-IPC returns notification that a port is no longer full (using an emergency message), the dispatcher removes from the send list the task that has been waiting the longest to send a message on this port, sends the message, and returns control to the task. If a task is not willing to block on a send, the message is queued internally and the task continues running. This is useful when a server wishes to present a consistent view of an object that might otherwise change (for example, a listing of the contents of a directory node). In either case, the use counts for ports in messages that aren't immediately sent are incremented to keep the ports from being inadvertently deallocated by other tasks.

Emergency messages are handled somewhat differently. We associate with each active port a list of emergency message handlers. Tasks can add and later remove emergency message handlers at the head of the list for a port of interest. When an emergency message is received on a port, the dispatcher starts at the front of the list and calls handlers in turn until one is willing to accept the emergency message. This approach, inspired by the nested exception handlers supported by Mesa, allows routines to easily specify special emergency message processing for certain sections of code without disturbing the handlers set up by higher levels.

Emergency messages reporting link failures require some extra handling. They come in on a special port (the dataport), are converted by the dispatcher to messages on the defunct port, and then passed to emergency message handlers for the port. If none of the emergency message handlers for the port accepts the message, the dispatcher checks to see if a receive is pending on the port. If so, the emergency message is converted to a normal message with ID FAILURE (in the format used by Roe) and passed to the task doing the receive. This both prevents receives from hanging forever because of lost connections and allows tasks to use the same error handling mechanisms

for both link failures and failure returns from other servers. Receives on an already dead port also return a FAILURE message. This handling of link failure emergency messages means that receives will always return a result, regardless of process or network failures (assuming, of course, that a result would have been returned in the absence of such failures). Using a non-blocking send in combination with such a receive provides, when needed, a predictable RPC facility.

Finally, a change was made in the way CMU-IPC allocates local capabilities. Instead of immediately reusing released capabilities, CMU-IPC now allocates them circularly. This change, which is transparent to user processes, makes it unlikely that a local capability will be reused while invalid references to it still exist. A more reliable solution would have been to associate a timestamp with each local capability, incrementing it each time the capability was reused and using it to validate capabilities before use. This was not worth the implementation effort and run time overhead involved.

The approach described above provides an environment that is considerably easier to deal with than the one currently used by Roe. Its central features are the simplification of send and receive primitives (without significant loss of functionality), the masking of the multiplexed nature of servers, flexible and unobtrusive handling of emergency messages, and uniform failure handling. While we have not converted Roe servers to use this approach (with the exception of incorporating emergency message handlers, send simplifications, and circular port allocation), we expect that it would both simplify servers and increase their reliability.

4.4.4. Towards a Better IPC

In the last section we presented mechanisms that would allow us to make effective use of the existing CMU-IPC implementation. In this section we make, based on our experiences implementing Roe, some recommendations for changes to CMU-IPC. Our intention here is not to address the basic philosophy of CMU-IPC, but rather to make relatively minor changes to the existing IPC that correct the shortcomings described in section 4.4.2. Our suggestions are in four areas: local capability generation, port allocation and deallocation, emergency messages and naming.

CMU-IPC reuses port names and local capability names when they are freed. As we have seen, this can lead to serious cases of mistaken identity. This could be avoided by making these names unique over the lifetime of a host or process. Unique names would allow obsolete port information to be easily detected, and would allow us to avoid the expensive invalidation process that is now done on port deallocation.

The mapping of ports to process local capabilities means that the name of a port depends on where it is being used. This allows a compact representation of sets of ports and avoids the problem of generating globally unique names. However, the complications this introduces when tracing and debugging a system more than offset the benefits. This could be avoided by either giving a port a common name across the network or by giving the user a way to translate a local name to a globally consistent one when needed.

Ports in CMU IPC are limited in number and must be explicitly allocated and deallocated. This leads to situations where ports are deallocated too soon or, for long-lived servers, ports are not deallocated soon enough (or at all) and the limit enforced by CMU-IPC is reached. Users of dynamically allocated memory will recognize this as being similar to the familiar "*memory leak*" problem, but complicated by the asynchronous nature of the environment. One problem is that a process has no way of finding out when there are no other processes holding references to a port. This could be solved by adding a reference count to a port, possibly with optional notification when the reference count drops to zero. Since CMU-IPC already keeps track of references to a port for security reasons, this is not a major modification.

Another approach would be to drop the need for dynamic port allocation and deallocation. One of the reasons for explicit allocation and deallocation is that the CMU-IPC kernel maintains internal state for each allocated port. Even if a port is not being referenced by others, this state is needed to perform the mapping from the local capability used by the process that owns the port to the internal port name. However, if the name were global, it would not be necessary to always maintain state for a port. In particular, if no other processes can refer to the port, there is no reason for the CMU-IPC kernel to know about it. Processes would still need to get a unique name for a port

when "creating" it, but no other action would be necessary until the port was shared. For shared ports, state would be needed for security and perhaps performance reasons.

CMU-IPC uses emergency messages to report link failures. These messages would be more useful if they provided information on the reason for the failure. For example, the distinction between port deallocation and host failure is an important one to some Roe servers. Next, delivering these messages through the defunct local capability rather than using a special port would allow a process to incorporate multiple packages that use CMU-IPC without worries of destructive interference. Finally, it is sometimes useful to know when an emergency message was sent relative to regular messages that are waiting to be received. One way of doing this would be to place a transparent mark in the incoming message stream (this is the approach used in the 4.2BSD IPC [Leffler 83]).

The string to port mapping facility maintained by CMU-IPC has two levels: 1) local, and 2) the rest of the network. There are situations in Roe where a more structured name space (perhaps broken up by subnet and host), combined with a facility for returning all occurrences of a given name, would have been useful. Another interesting possibility would be to allow parameters ("location," "operating system," and so on) to be associated with a name lookup request [Bukys 83].

4.4.5. Other Difficulties

Two other problems we encountered were: 1) inadequate file system support on some hosts, and 2) lack of distributed debugging facilities.

Roe requires some minimal file system support in order to implement atomic file operations on hosts that store replicated files, directories, and files that will be accessed atomically. In practice this means that there must be a way to ensure that data written to a file is actually on disk and is likely to survive a machine crash. UNIX and the Alto/Mesa environment provided this facility. RIG did not. RIG also did not provide information on the free disk space on a machine and both RIG and Alto/Mesa had no information on the level of activity on a machine. The shortcomings of RIG would have prevented it from being a full member of the Roe community if we had ever

implemented crash recovery. Plans were made to add some of the needed features to the RIG file system, but this was never done.

One major surprise was the degree of difficulty distributing a system adds to the debugging process. The lack of a debugger that would work across multiple processes, combined with CMU-IPC's anonymous ports, timing problems, and race conditions present in an asynchronous environment, led to situations where it sometimes took *days* to track down even fairly simple problems.

Our general approach to debugging Roe was to have each server generate a log of all messages sent and received. These logs were then combined to produce an overall trace of the activity in the system. There were problems with this approach. The use of local port capabilities by CMU-IPC made it difficult to correlate the logs. The message-by-message logs were too "detail intensive" for many applications. Finally, this approach didn't give us any control over the relative timing of operations. In fact, turning on logging sometimes affected the timing enough that problems we were trying to trace disappeared.

Based on our experience, it is clear that a distributed debugger is essential when implementing a distributed application of any size. At a minimum, such a debugger should be able to trace the message flow in a system. Other desirable features include the ability to examine and modify messages in transit, to control the relative timing of events, and facilities for debugging a process in isolation, using synthetic or previously recorded message traces. Spider [Smith 81c] and Idd [Harter 85] are two distributed debuggers that provide many of these facilities but were, unfortunately, unavailable for this implementation effort. Taking a somewhat more sophisticated approach, it would be useful to be able to retain a high degree of control over individual processes (including single step, variable examination and so on), while allowing grouping and abstraction where needed. For example, if the code to migrate files is being tested, there is generally no need to see the details of quorum collection. Finally, when debugging Roe, it would have been very useful to be able to place constraints on the order of transmission of messages and then to have the system run with unconstrained times being varied. This could have helped us track down a number of timing problems.

4.5. Performance Considerations

In the following sections we present some rough performance figures for Roe. It should be pointed out that both Roe and the IPC it uses are prototypes and have not been optimized for performance. Despite this, it is illuminating to look at where the time performing Roe operations is spent. We will use for our example opening an existing Roe file on a UNIX host. The next section presents some timing figures for basic operations in the host environment. Section 4.5.2 uses these figures, along with measurements of Roe, to provide a breakdown of the time required to open a Roe file. Section 4.5.3 makes some suggestions for improvements.

4.5.1. Host Performance

Our measurements were done on VAX[4] 11/750s with 2MB of memory, Fujitsu M2351A/AF Winchester disk drives (18ms average seek; 7.5ms latency), connected by a 3MB Ethernet. The systems were running 4.2BSD UNIX in multiuser mode. They were idle except for normal background activity (*rwhod*, *cron* and the like). All of the results given below are *elapsed* times in milliseconds. Timing was done using the software clock maintained by UNIX. This clock has a resolution of 10ms, and so we generally timed repeated calls to improve the effective resolution. We ran 10 sets of measurements for each call or sequence of calls being tested and then calculated the average and standard deviation.

sequence	msec/sequence	description
fopen/fclose	6.4±0.3	open/close file; single path component; inode cached
first fread	90±11	read 1024 bytes from an opened file (first read)
subsequent freads	32±11	repeatedly reading 1024 bytes from an opened file
20*getw+100*getc	6±2	reading a small property list (after first fread)
ftell/rewind/fseek	1.6±0.2	file positioning

Table 4-1: File access times for 4.2BSD UNIX

[4]VAX is a trademark of Digital Equipment Corporation.

Table 4-1 shows 4.2BSD UNIX file system access times for some operations commonly used by Roe. The value given for fopen/fclose is for opening an existing file in the current directory when its inode is already cached. This is the most common type of open performed by Roe.

In Table 4-2 we have listed some times for message and port management. Note the relatively high cost of releasing a port. Most of the cost may be attributed to CMU-IPC's attempts to locate and invalidate any existing references to the port. Since there was no other IPC activity on the system when this measurement was done (and so no outstanding messages), this figure is a minimum. The cost for remote name lookup is also surprisingly high. The user code implementation of the network portion of the IPC requires a number of context switches to send the lower level packets and acknowledgements used to reliably send a message across the network (in this case, the reply to the broadcast request). There is clearly substantial room for improvement here.

Table 4-3 shows the elapsed time required to send a simple message and have it received by another process. In all cases the time was found by measuring the round trip time for a message and then halving to find the one-way time. In the case where there are two destinations, the

operation	msec/operation	description
new_port	2.3±0.3	allocate a port and increment its use count
release_port	15.7±1.3	decrement use count and deallocate port
message formatting	2.6±0.1	typical message formatting sequence
local name lookup	2.6±0.4	look up port associated with local name
remote name lookup	190±30	look up port associated with remote name

Table 4-2: Message and port management costs

destination	message contents		
	empty	512 bytes	1 port
local	4.9±0.2	6.8±0.3	7.4±0.7
remote	100±4	143±7	117±7
2 remote	175±10	190±11	170±11

Table 4-3: Time to send a simple message

sending process sent both messages and then waited for replies (CMU-IPC doesn't have a multi-cast capability and so Roe is also forced to take this approach). The most striking result in this table is the poor performance of the network IPC. Sending a message to a remote process takes 20 times as long as sending one to a local process. A remote procedure call using this IPC would take 200ms. For comparison purposes, an RPC using 4.2BSD UDP on similar hardware takes 26.5ms [Cooper 85] and an RPC in the V system takes 2.54ms on a 10MHz Sun workstation [Cheriton 83]. These figures tell us that in our environment IPC costs are likely to dominate for any operation in Roe that requires access to remote resources, but that this is not necessarily the case in other environments. We will have more to say on this in the next section.

4.5.2. The Performance of Roe

In this section we analyze the cost (in terms of time delay) of opening a Roe file. Opening a file is one of the most complicated and frequently used operations in Roe, and so the time required to open a file determines, to a large extent, the overall performance of Roe. We consider in detail only the case of opening an existing unreplicated file. The message traffic for this case in the current implementation was shown earlier (Figure 4-6). We assume that all resources and servers are local and that connections exist to all servers we will be using. Times for remote cases may be found by adding the appropriate cost for each message shown in Figure 4-6 that crosses a machine boundary.

phase	estimated time (ms)	fraction	actual time (ms)
send/accept request	8.0	3%	est
read directory (LDS)	26.3	10%	28±3
open file (LFS)	154.1	59%	135±36
GDS overhead	31.1	12%	est
return result (TC)	41.0	16%	est
total	260.5	100%	268±32

Table 4-4: Time to open an existing local Roe file, by phase of open

Table 4-4 shows the elapsed time required to open a Roe file, broken down by the phases of the open. The estimates shown here were done by counting calls made by servers to the various routines shown in earlier tables. These estimates agree with the actual measurements given in the table. Note that the actual open of the file copy (managed by the Local File Server) takes almost 2/3 of the time. The remainder of the time is fairly evenly divided between reading the directory entry of the file, returning the result to the user (through the Transaction Coordinator), and internal Global Directory Server overhead.

The time to open a file is broken down by types of activity in Table 4-5. Note that IPC related activity (message send/receive, message formatting, and port management) accounts for half of the total cost. Most of the rest of the cost is due to reading the property list of the file copy.

activity	estimated time (ms)	fraction
message send/receive	53	20%
message formatting	19	7%
port management	64	24%
table/list search	10	4%
file open	6	2%
read file properties	105	39%
other	11	4%
total	268±32	100%

Table 4-5: Time to open an existing local Roe file, by type of activity

copy locations	estimated time (ms)	fraction of local
1 local	261	1.0
1 remote	490	1.9
2 remote	600	2.3
1 local, 2 remote	610	2.3
3 remote	710	2.7
5 remote	930	3.6

Table 4-6: Opening remote and replicated Roe files

The Roe file open time of 268ms for local files is not very impressive when compared to the measured UNIX time of 6.4ms. To be fair, Roe also reads in the first block of a file and so the equivalent UNIX time is 96ms, making a Roe open of a local file roughly 3 times slower than the equivalent UNIX open. If the file is remote or replicated, the Roe open is considerably slower. Table 4-6 shows estimates of the cost to open remote and replicated Roe files. Rough measurements (where possible) confirm these figures. The drastic increases in open times for the remote and replicated cases are due to the poor performance of our network IPC (Table 4-3). For the case of a single remote copy, IPC overhead accounts for 3/4 of the file open overhead.

4.5.3. Improvements

There are a number of changes that could be made to the Roe implementation to improve its performance. The following changes offer the possibility of significant performance improvements without compromising the transparency goals of Roe:

- Preallocation of ports. A quarter of the time required to open a local Roe file is spent managing ports. *Nearly all of this time is used allocating and deallocating ports* used to represent short-lived objects and requests. If servers preallocated ports and then managed these ports internally, this overhead could be avoided.
- Consolidation of servers. Another quarter of the time used opening a local Roe file is spent sending messages between local servers and to and from the users. Consolidating local servers into fewer processes would decrease the number of local messages sent. Consolidating all Roe servers on a machine into one process would reduce the message passing costs by 70%. Placing Roe in the kernel would eliminate message costs for local file opens.
- Advisory locks. The two changes described above would cut in half the time required to open a local Roe file in the current implementation. If these changes are made, 85% of the remaining time would be spent actually opening a file and reading its property list to get voting information. This overhead could be avoided in many cases by placing *advisory locks* on frequently used files. An advisory lock would work as follows:

If a GDS believes that a copy of an opened file will be used again shortly, it would specify, when the file was closed, that the LFS not actually close the file but rather demote the read or write lock on the file to an advisory lock. If the file copy is opened for write by another GDS, this lock is "broken" and the GDS is advised that the copy is not available. If a GDS holding an advisory lock on a file copy wishes to open it for write, it asks the LFS managing the copy to change the advisory lock into a write lock. If opening for read, the GDS can either change the lock to a read lock or use the advisory lock. In either case, since the copy's data and properties have not changed, there is no need to explicitly collect a quorum again and locally retained information could be used. This is a particularly effective technique when a file is replicated, with one copy being local and the others remote. In this situation, the advisory locks could be used to eliminate remote operations for read access.

- File and directory caching. There are a number of techniques for maintaining consistent distributed caches of information (see, for example, [Archibald 84, Sheltzer 86]). These techniques could be used in addition to or in place of advisory locks in Roe. Since we expect a high degree of locality in both file and directory references and a high proportion of read references (see Chapters 5 and 6), local caches of file and directory information should result in a significant decrease in network activity. Note that it is necessary to keep local caches consistent, since the design of Roe doesn't allow for "best guess" hints here.
- IPC performance improvements. There is significant room for improvement in the performance of our network IPC. A kernel implementation followed by careful tuning would be, at least at the application level, the least disruptive approach. An alternative would be to consider the use of a simple RPC mechanism that would lend itself more readily to a high performance kernel implementation (see, for example, the V kernel RPC [Cheriton 83]).
- Lower level host system interfaces. The current Roe implementation uses existing host file systems. This adds some layers of software between Roe and its eventual destination (the disk). Using a lower level interface to these file systems (e.g., *read* instead of

fread on UNIX) or ignoring the host file systems altogether and implementing Roe on the raw disk would decrease software overhead. Unfortunately, the implementation costs of the second approach can be quite high, particularly in a heterogeneous environment.

4.6. Observations

We can make the following observations about Roe based on our implementation experiences:

- The approach used by Roe is a powerful means of accommodating the heterogeneity present in a network. Minimal host changes allow new types of hosts to be easily added. The combination of local servers with a uniform interface, typed file data, and automatic type conversion at machine boundaries allows heterogeneity to be ignored where it is not important. Machine-dependent files, property "escapes," and the network model maintained by Roe provide mechanisms for recognizing situations where heterogeneity is important and for exploiting it when possible.
- The modular structure of Roe makes it easy to partially integrate new hosts. Functionality can be added as needed to increase performance and availability. This modular structure also helps reduce complexity by providing well-defined interfaces and simple abstractions.
- The underlying host file system support expected by Roe is not always present. At a minimum, Roe requires a means of insuring that information written to a file will survive a machine crash (usually this means insuring that it has reached the disk). In addition, information on free space and congestion is useful when making placement and migration decisions. It is also worth noting that the Roe file access protocol assumes that files are represented and used as seekable streams. Insuring that operations survive a machine crash is useful outside the context of Roe and so this expectation is not unreasonable. Status information can be approximated (or maintained by local servers if Roe is the only user of a file system). Record and other file abstractions can be implemented using SEEK and properties (although with some loss of efficiency).

- Properties are a surprisingly useful concept. Properties were originally introduced in Roe to provide a means for users to associate additional information with their files [Bukys 82]. However, they were quickly recognized as a convenient way of associating Roe-specific information with files, as a flexible method for specifying arbitrary parameters in LOGIN, OPEN and other requests, and as a way to specify host-specific file operations.
- A system that provides transaction support should provide facilities for transactions that span multiple objects. This is convenient both for the user and the implementor of higher levels of the system. In the case of Roe, this would have made implementation of the Global Directory Server considerably easier.
- The lack of need for global knowledge simplified the Roe implementation. In particular, we were able to avoid complicated and potentially expensive global consistency and snapshot algorithms. Roe maintains a partial view of the network, with status changes propagated only where they are likely to be needed. Information on objects managed by Roe is kept with the object and verified only when the object is used.
- Finally, and perhaps most important, our implementation validates the design presented in Chapter 3. The Roe prototype implementation shows that it is both possible and practical to provide a fully transparent distributed file system for a heterogeneous local area network. In addition to providing transparent access to replicated and distributed heterogeneous resources, Roe is able to transparently reconfigure to adapt to losses and additions to the network and to adjust to changing usage patterns. Roe scales well as the size of a LAN increases. These factors make Roe a powerful tool for effectively using the resources of a network, without the need to know the details of these resources.

4.7. Summary

In this chapter we have described the Roe prototype implementation. The implementation runs on a heterogeneous local area network of UNIX, RIG, and Alto/Mesa hosts. It supports a distributed

and replicated global directory (on UNIX hosts), replicated files (on all hosts), automatic file placement, and simple demand-based migration. The system provides fully transparent access to its distributed and replicated resources.

Implementing the Roe prototype turned out to be surprisingly difficult. Although the structure of Roe and the algorithms used were straightforward, the approaches and tools we used greatly complicated matters. The majority of the implementation was done in C using multiplexed servers and an asynchronous message-based IPC. It is clear in retrospect that a project such as this should probably not be attempted without appropriate language support and a distributed debugger. Using remote procedure call as the basic IPC mechanism would have also avoided several problems.

The Roe implementation meets the transparency, availability, heterogeneity, reconfigurability, and scalability goals described in Chapter 3. The only major failing of the implementation is in the area of performance. Many of the performance problems may be attributed to the prototype nature of both Roe and the IPC it uses. Reimplementation and tuning would correct these problems. Our use of remote, replicated, and distributed resources, along with the insistence on consistency, also had a negative impact on performance. Techniques such as advisory locking and caching could be used to minimize this impact.

Our experiences with the Roe implementation show that a fully transparent distributed file system for a heterogeneous LAN is achievable. The high degree of transparency supported by Roe lets the user ignore the presence of the underlying network and allows Roe to respond to changing demands and resources. These benefits provide compelling justification for the approach used by Roe.

Chapter 5

Short-Term File Reference Patterns in a UNIX Environment

5.1. Introduction

Roe provides users with a distributed, hierarchically structured file system similar in appearance to the UNIX[1] file system. A number of other DFSs have also adapted this model [Lyon 85, Satyanarayanan 85, Tichy 84, Walker 83b]. Understanding the benefits and drawbacks of the approaches used by each of these systems has been hampered by a lack of information on the ways in which they are used. In particular, there is little data available on short-term file reference patterns, and no data at all on directory usage, despite the observation by several researchers that name resolution appears to account for up to half of the activity in systems that they have studied [Leffler 84, Ousterhout 85, Sheltzer 85].

Data on file and directory reference patterns can be used to evaluate the performance of existing DFSs, and as an aid in identifying and correcting problems. It can also be used as a guide in developing new DFSs by providing information on key areas such as name lookup overhead,

[1]UNIX is a trademark of AT&T Bell Laboratories.

read/write ratios, interreference intervals, data lifetime, and sharing. Traces of reference activity can be used as an input to simulations that evaluate DFS design, or examine individual issues that arise in designing DFSs. Examples include developing algorithms for file and directory placement, migration, update propagation, and investigating the overhead introduced by replication.

Inspired by these benefits and frustrated by the lack of information, we instrumented a local UNIX system and collected information on file system requests. The UNIX file system is particularly appropriate for this study. It places relatively few constraints on user behavior and has been used as the design model for many recent DFSs.

This chapter describes the data collection method and presents our analysis of short term file reference patterns. Chapter 6 presents an analysis of short term directory reference patterns. We have generally tried to present results in a way that gives a qualitative feel for the characteristics of the data we have measured. Quantitative fits and distributions are, for the most part, sacrificed in favor of observations that would aid in developing and operating DFSs.

The work described in these two chapters is novel in several respects. It is by far the most detailed and comprehensive study of short term UNIX file reference patterns that has been done to date. It provides the only results we have seen on directory reference patterns. It is also the only study we have seen that examines the differences between important user and file classes. In addition to examining the overall request behavior, we have broken down references by the type of file, directory, and requestor. We see large differences in behavior between the various classes. Knowledge of these differences should be useful in designing future DFSs.

Section 5.2 describes the environment in which our measurements were made. Sections 5.3 and 5.4 present an overview of the data collection and analysis methods. In section 5.5, we present some of the results of the analysis. A summary of our results is presented in section 5.6

Familiarity with UNIX [Ritchie 78] is assumed. Knowledge of 4.2BSD UNIX [Joy 83] may also be useful.

5.2. Data Collection Environment

The data used here were collected from a VAX 11/780 on the University of Rochester Computer Science Department Internet. At the time that the data were collected (September 1985), the internet consisted of a VAX[2] 11/780, 4 VAX 11/750's, 7 Sun workstations, 13 Xerox Dandelion workstations, 3 Symbolics LISP machines and a number of special purpose devices. The 11/780, Seneca[3], was selected as the primary machine for data collection because it was by far the most heavily used of our systems. Seneca had, at the time, 4MB of memory, 560MB of disk storage and was running 4.2BSD UNIX. The system supported roughly 200 users. The primary user activities were program development (as part of our research effort), text editing and formatting, reading news and reading personal mail. Seneca also acted as a USENET news and UUCP mail relay [Nowitz 78]. There was relatively little database activity.

Data were also collected from two of the 11/750's. Preliminary analysis of the 11/750 data merely confirmed the importance of Seneca in our environment. Neither of the 11/750's had file system activity levels greater than 15% of that seen on Seneca. Because of this, only the Seneca data were fully analyzed.

5.3. Data Collection Method

Two types of data were collected: 1) a static "snapshot" of the file system, and 2) a running log of file system activity.

5.3.1. Static Snapshot

The static snapshot provides a picture of the entire file structure on a machine at a given point in time. The information generated for each file system object that we are interested in is given in Table 5-1. Processing starts at the root of the file system hierarchy and recursively traverses the directory tree, logging each object encountered.

[2]VAX is a trademark of Digital Equipment Corporation.

A static snapshot was taken of the Seneca file system when file system logging (section 5.3.2) was started. This snapshot was used as a starting point for the analysis programs (section 5.4) and also provided information on the static file size distribution.

5.3.2. Logging File System Activity

The 4.2BSD UNIX kernel was modified to log selected system calls made by users. The calls logged can be classified as follows:

(1) Directory structure modifications: mkdir, rename, rmdir and symlink.

(2) Process context: chdir, chroot, exit, fork/vfork and setreuid.

(3) Other references: close, execv/execve, link, open/creat, truncate, unlink.

The logging of these calls has a negligible effect on the performance of the host (less than 1%).

A number of other file-system related calls were judged unnecessary for our purposes (due to our ability to infer them from other calls or to their infrequent use) and were ignored. These included:

(1) Internal file operations: read, write, lseek. Actually, code was added to log reads, writes and seeks. However, running with this code enabled increased the size of log files by 500% and resulted in a 5-10% degradation in host performance. Since we were concerned primarily with operations on files as a whole, this additional overhead was unacceptable. Instead, we summarized some of the information in close records

object	output
directory	name, device, inode
regular file	name, device, inode, size (bytes)
symbolic link	name, target file
special file	name

Table 5-1: snapshot output

[3]Our local VAXen are named after Western New York State's Finger Lakes.

(see Table 5-2).

(2) Protection calls: chmod, fchmod, chown, fchown.

(3) Status calls: readlink, fstat, lstat, stat, utimes, access.

(4) Other calls: fcntl, flock, fsync, mknod, ftruncate.

Each log record included the time that the call finished (with a resolution of 10ms) and the pid (process identifier) of the process making the request. In addition, most records contained information describing the call arguments and result. The record contents are given in Table 5-2.

A brief explanation of the contents of Table 5-2 is in order at this point. The first four records (mkdir, rename, rmdir and symlink), combined with the results of a static snapshot taken at the start of logging, allow us to construct and maintain a model of the directory tree for the file systems on the machine. *Mkdir* creates a new directory. *Rename* changes the path used to reach an object. *Rmdir* deletes a directory. *Symlink* creates a symbolic link containing a path to a file or directory. When a symbolic link is encountered during path resolution, the path in the symbolic

call	output
<all>	time, pid of caller
mkdir	+ file id of new directory, path of new directory
rename	+ old path, new path
rmdir	+ path of deleted directory
symlink	+ target of link, link name
chdir	+ path to new working directory
chroot	+ path of new root
exit	-
fork (fork/vfork)	+ child pid
setreuid	+ new ruid
close	+ file id, final size, bytes read, bytes written
execute (execv/execve)	+ file id, uid of file owner, size, path
link	+ target path, link path
open (open/creat)	+ file id, open flags, mode of file, size, uid of file owner, path
truncate	+ path, new size
unlink	+ path

Table 5-2: dynamic log structure

link is substituted into the partially resolved path before resolution is continued. This is the only way in 4.2BSD UNIX to make links across file systems.

The next 5 records (chdir, chroot, exit, fork and setreuid) give us the information we need to keep track of the working directory and real uid (ruid) of each process. *Chdir* changes the directory used to resolve relative references made by a process (those not starting from the root of the file system tree). *Chroot* changes the root of the file system as seen by a process. *Fork* (fork and vfork system calls) and *exit* create and destroy processes. Logging these allows us to keep track of processes created for each user. *Setreuid* changes the effective "owner" of the current process. This is the mechanism for logging into the system.

The remaining records (close, execute, link, open, truncate and unlink) are the actual references to files. *Execute* (execv and execve system calls) executes a file, replacing the current process with the image given in the file. *Link* and *unlink* add and delete directory entries for files. If unlink removes the last link to a file, the file is deleted. *Open* (open and creat system calls) opens or creates a file or opens a directory. Processes access files either by explicitly opening them or by inheriting open files from their parents. *Truncate* shortens a file. *Close* records indicate that a process no longer has a file open. They are generated by either a close system call or by a process exit. As mentioned earlier, a process may inherit open files from its parent. If this happens, the close record is generated when the last process having access to a file due to the open closes the file or exits (for those in the know: we log the release of the kernel open file table entry). We only log closes for regular (data) files. Closes are not logged for directories or for special files (files corresponding to devices). Since directories are short, completely scanned when opened and can only be opened for reading, close records for directory opens would have given us little useful information. Special files are not analyzed in this study (except for a count of opens).

The calls listed in Table 5-2 are logged for all processes in the system. In addition, a small number of administrative records having to do with enabling and disabling data collection are logged. The most important of these is the *process state* record. A process state record contains information on the ruid, working directory, root directory and command name for a process. One

of these is logged for a process the first time it appears in a log, but only if we don't already have this information for the process. Process state records are only necessary for processes that exist before logging is started (and for their children until we log the parent). They give us a way to locate the process in the directory tree and to classify it as a user, system or net process.

The 4.2BSD tracing package we have described differs from the one developed independently at Berkeley by Zhou et al. [Zhou 85] in a number of ways. The most important difference is that we don't collect information on internal file operations. This means that we have less information on the timing of these operations to files and on which bytes are accessed. We do, however, log the number of bytes read from or written to an opened file. As we will see later on (section 5.5.1), most files in our environment are read or written completely and are usually open for only a short period of time. These results, combined with the fact that most DFSs treat files as a whole, mean that the omission of internal file operations is not important for our particular application.

We also collect less information per record. In particular, all of our times are real times at the finish of the system call. Zhou et al. record, in addition to real times, the duration of the call and process virtual times. We made a decision early on to collect the minimum information necessary for our purposes. This allows us to collect and process data for a longer period, but means that our trace is sensitive to the capacity of the machine that the data were collected on. Adjusting for this would be difficult in any case.

Finally, we collect information on high level directory operations (create, delete and open). This allows us to track process locations in the directory tree so that we can accurately analyze relative file references. It also gives us the data needed to analyze directory reference patterns.

The trace data collected by Ousterhout et al. [Ousterhout 85] includes information on seeks (so that read and write data may be derived), but lacks information, which we record, that allows references to be classified by file type and file owner. We also include directory and process information not present in their trace.

Note that our package does *not* collect a full trace of file system activity. We don't collect information on inode accesses, paging activity, internal file operations (except for the total number of bytes read and written), or protection and status related calls. However, our package does generate detailed information on the most common operations on files and directories as a whole (open, close, create, delete, execute and so on) and on overall read and write activity for opened files. This information provides a useful basis for investigating file usage patterns and is sufficient for trace driven studies of most DFSs.

5.4. Analysis Method

5.4.1. Basic Approach

The data in the raw form described in Table 5-2 are difficult to analyze. There is no obvious correspondence between opens and closes, unlinks are not associated with the files they affect, no direct information is available on process working directory or owner and so on. A library of analysis routines was written to address these difficulties. The routines maintain enough state about the system being analyzed to allow the necessary associations to be made. Alternatives would have been to reformat the file reference logs so that each record contained the information necessary for its analysis (see, for example [Zhou 85]) or to collect more information for each reference. We chose to derive the information at the time of analysis to minimize the disk resources needed (and so maximize the logging period). Of course, one pays a penalty in analysis time for doing this. Using this approach, a simple analysis of the trace described in this chapter (2.5 million events occupying 70MB of disk) takes about 5 hours of 11/780 CPU time. This is adequately fast for our needs.

Analysis proceeds in two phases. During the initialization phase, a snapshot of the directories in the system being analyzed is read in and used to set up a model of the original directory structure. During data analysis, log records are read and passed, one by one, to user analysis routines. These log records are also used to update state information on files, directories and processes in the system, creating and destroying them to maintain an accurate model. Given this up-to-date state

information, the library routines can perform the associations mentioned above and pass this information on to the user routines.

There are some conventions worth mentioning here that are used by all analysis programs:

(1) Calculations involving file sizes are always based on the size of the file when it is closed or executed.

(2) File reads and writes are assumed to occur at the time a file is closed (we didn't have more accurate information on these operations). Since the time most files are open is usually considerably shorter than any of our histogram resolutions, this has no noticeable effect on our results.

(3) File lifetimes run from the time a file is created (based on a create flag in the open call) until the time the underlying inode supporting the file is deleted. This doesn't happen until there are no links to the file left *and* there are no active opens, so the delete time can (and frequently does) differ from the time of the last unlink. File version lifetimes are handled in a similar fashion.

(4) Processes occasionally open a file and then open it again before closing it. This is usually done to get both read and write access to a file without using the mechanisms for this built into the 4.2BSD kernel. We honor the intent (not the method) by combining these opens into one open with both access modes. This affects only 0.7% of the opens and so is not an important consideration in any case.

5.4.2. Cuts

We are interested in investigating both the overall pattern of requests to the file system and in the patterns for various classes of users and files. Past work has often ignored the distinction between batch and interactive use, system and user files, log and permanent files and so on. We believe that information on the behavior of each of these file and user classes can be of great value in developing a DFS and have developed a number of data cuts to separate the classes of interest. We use three basic types of cuts:

(1) Cuts on the ruid (owner) of processes making requests (UUCP/USENET network, system and user).

(2) Cuts on the owner of files (UUCP/USENET network, system and user).

(3) Cuts based on the purpose of files (log, permanent, temporary).

Some of these can be combined to give other more specific cuts. 14 cuts are used in this analysis. The cuts and their meanings are:

(1) **no cut**: This cut passes all records in the log to the user analysis routines.

(2) **ruid_NET**: Passes references by what we term *net* processes. Net processes are those running under UUCP, USENET news or notes accounts. Most of these processes run in batch mode and so this cut gives us a sample that is considerably different from an interactive one. This category has been broken out from the system and user categories because of the batch-oriented nature of the references and the large number of references by net processes (roughly 1/3 of the references in this study and as much as 70% of the non-system references in earlier studies [Floyd 85]). We don't include references due to Seneca being on the Rochester Internet in the ruid_NET category.

(3) **ruid_SYSTEM**: Passes references by system processes (those running under root, daemon, games and other miscellaneous system accounts). System processes are primarily daemons that provided widely used services (such as spooling and network status reporting), processes created on behalf of users to perform privileged operations, and periodic maintenance processes.

(4) **ruid_USER**: Passes references by processes running under user accounts.

(5) **owner_NET**: Passes references to files owned by UUCP, USENET news and notes accounts. These are primarily news articles and UUCP spool files.

(6) **owner_SYSTEM**: Passes references to files owned by the system accounts mentioned above. This includes major administrative and status files (for example, /etc/passwd), system libraries, system include files and so on.

(7) **owner_USER**: Passes references to user files.

(8) **file_LOG**: A number of files on any UNIX system are used to keep logs of activity. Examples include /usr/adm/messages, /usr/adm/wtmp and user mbox files. Since we expect the access patterns for these files to be considerably different from that for files as a whole and since these files are generally quite large, we use a cut, file_LOG, that allows us to analyze only these logs.

We had originally intended to place in this category just those files opened with append-only access. However, it soon became clear that this mode of access is basically never used. Instead, most logs are opened write-only, a seek is done to the end of the file and then the log entry is appended. If several processes are trying to update a log simultaneously, the results are unpredictable. Some of the busier logs on our system are scrambled on a regular basis using this "method."

We were eventually forced to use the name of the file given in the open call to make this cut. Luckily, most of the log files on the system have well known names and an examination of the sources for commonly run programs and of the file reference logs enabled us to find the rest of the log files on the system.

(9) **file_PERM**: Passes references to permanent files. This includes all files that aren't log files (file_LOG) or temporary files (file_TEMP).

(10) **file_TEMP**: Passes references to temporary files. This includes files that are created on a special file system (/tmp), temporary spool files, lock files and other such transitory files. Most temp files are clearly identified by either their name (a special template is usually used to create temp file names) or by the directory in which they are created.

(11) **owner_USER+ruid_USER** (shown as U in tables and figures): Passes references that satisfy both the owner_USER and ruid_USER cuts. These are user references to user files. The owner_USER+ruid_USER cut produces results similar to the owner_USER cut. There are about 9.5% fewer references for the U cut, but the resultant distributions are nearly identical. It is included here for comparison with the next three cuts.

(12) **owner_USER+ruid_USER+file_LOG** (shown as **U+file_LOG** in tables and figures): Passes user references to user log files.

(13) **owner_USER+ruid_USER+file_PERM** (shown as **U+file_PERM** in tables and figures): Passes user references to user permanent files.

(14) **owner_USER+ruid_USER+file_TEMP** (shown as **U+file_TEMP** in tables and figures): Passes user references to user temporary files.

5.4.3. Analysis Compli ations

The data analysis did not proceed as smoothly as we had hoped. This section describes some of the problems we experienced and suggests changes in the data collection and analysis that would help avoid these problems in the future. None of these problems was serious enough to have a noticeable effect on our results.

It was not always possible to pair up opens and closes correctly. In most cases there was only one open for a given file to associate a close with, or the process numbers of an open and close matched. In cases where this was not true, we looked for an open that was made by an ancestor of the process making the close request. Sometimes there were multiple opens to a file outstanding among those made by ancestors. This problem occurred less than 0.03% of the time and was dealt with by using the most recent open by an ancestor. A more accurate solution would have been to record an open session number in the log and use this to make the association, but the low frequency of occurrence of this problem and the relative unimportance of the derived numbers make this solution unnecessary for us.

There were two peculiarities in the 4.2BSD kernel that resulted in some surprises in the logs. One, having to do with incorrect returns from the fork call, was caught and corrected before the data analyzed here were collected. The other, an inconsistent handling of error indicators when a process was forcibly terminated, caused us to lose some close and exit records. This was not discovered until fairly late in the analysis. Less than 0.03% of the close records and about 0.5% of the exit records were not recorded because of this problem. Since the number of close records

lost was so low, we made no attempt to correct the problem.

Files were classified (as log, perm or temp) the first time they were seen in the logs. Occasionally this classification was incorrect. While we were developing the cuts, we classified a number of files by hand, using information on the programs making the requests and the full history of references to the files. Comparing our classifications to those done by the analysis routines (using file name and directory information) showed that only a few tenths of a percent of files were incorrectly classified. It would be difficult to do better than this without explicit information on the intended usage of files. This information is just not available under UNIX.

We had to retain a large amount of state in order to associate unlink records with files and to interpret their meaning. Since we needed most of this state for other reasons (uid classification, directory studies and so on) this was not really a problem for us. Including the file id and a count of the number of remaining references in unlink records would make it possible to interpret them in the absence of the state information.

5.5. File Reference Patterns

Roughly 7 days of data were collected on Seneca (168.82 hours, from 3:21am on Monday, September 16, 1985 to 4:10am on Monday, September 23). During this period there were 142 active users of the system. There were generally 20 to 30 logged-in users at any given time on weekday afternoons, with load averages running between 5 and 10.

In section 5.5.1, we examine the overall pattern of open and read/write requests. Section 5.5.2 briefly examines execve patterns. Section 5.5.3 concentrates on user files. Our approach in all cases is to present only those tables and histograms that are particularly characteristic or striking.

5.5.1. Overall Open and Read/Write Patterns

5.5.1.1. Basic Statistics

A summary of the records collected is given in Table 5-3. The first three columns give the number of records of each type collected, the average rate for that type of record, and the percentage of the collected records that this represents. The remaining columns show the number of records collected cut by the ruid of the calling process and the percentage of the total for the ruid class.

From this table we can see that each of our ruid categories accounted for roughly 1/3 of the activity on the system. The majority of the file system requests were for opens and closes, with most of the rest of the categories being a factor of 5 or more down from this (of course we didn't record reads, writes and seeks, all of which would be a significant component of a full trace). Processes made, on the average, 5.3 open requests.

record	no cut			ruid_NET		ruid_SYSTEM		ruid_USER	
	count	per hr	fraction	count	fraction	count	fraction	count	fraction
mkdir	936	5.5	0.04%	795	0.11%	2	0%	139	0.02%
rename	3211	19	0.13%	1946	0.26%	408	0.04%	857	0.11%
rmdir	913	5.4	0.04%	780	0.11%	0	-	133	0.02%
symlink	16	0.1	0%	0	-	3	0%	13	0%
chdir	136063	806	5.4%	19102	2.6%	71854	7.1%	45106	5.7%
chroot	0	-	-	0	-	0	-	0	-
exit	180270	1070	7.1%	31219	4.2%	85917	8.5%	63133	8.0%
fork	181511	1080	7.1%	29271	4.0%	90735	8.9%	61503	7.8%
setreuid	16772	99	0.66%	4372	0.59%	9698	0.95%	2701	0.34%
close	754072	4470	29.7%	249837	34.0%	298164	29.4%	205666	26.2%
execute	125064	741	4.9%	26761	3.6%	38093	3.8%	60209	7.7%
link	42929	254	1.7%	25694	3.5%	7301	0.72%	9934	1.3%
open	965087	5720	38.0%	277350	37.7%	393661	38.8%	294070	37.4%
truncate	0	-	-	0	-	0	-	0	-
unlink	130929	776	5.2%	68342	9.3%	19861	2.0%	42726	5.4%
total	2537773	15040	100%	735469	100%	1015697	100%	786190	100%

Table 5-3: records logged

Table 5-4 gives the number of opens to each type of file object on the system. For the purposes of comparison, the SLAC trace [Porcar 82] included about 237,000 opens to data (regular) files in a similar period. The remainder of the analysis in this chapter deals with only regular files, the largest category in Table 5-4. Directory access patterns (including explicit directory opens) are analyzed in Chapter 6. Block and character special files are used in UNIX to provide access to devices and are not of interest to us. They are, in any case, a small fraction of the total number of opens.

Opens may be further broken down by the class of file being opened and by the owner of the file. This information, plus statistics on how many files there are in each category, is given in Table 5-5. We see here that 2/3 of the references were to perm files, although temp files made up 4/5 of the files referenced. Relatively few references were made to user files. The large number of net files may be attributed to a daily news expiration procedure that reads the headers of all news articles.

Information on read/write modes for open-close sessions is given in Table 5-6 (note that percentages in this table sum horizontally). Overall, files opens were evenly split between opens with read-only access and opens for write-only or read-write. Users, however, opened most files read-

type	no cu.		ruid_NET		ruid_SYSTEM		ruid_USER	
	opens	fraction	opens	fraction	opens	fraction	opens	fraction
regular file	754285	78.2%	249825	90.1%	298186	75.7%	206268	70.1%
directory	170448	17.7%	17275	6.2%	72625	18.4%	80548	27.4%
block special	922	0.1%	0	-	60	0.02%	862	0.3%
character special	39432	4.1%	10250	3.7%	22790	5.8%	6392	2.2%
total	965087	100%	277350	100%	393661	100%	294070	100%

Table 5-4: Opens, by object type

cut	opens	% opens	files	% files	opens/file
file_LOG	35662	4.7%	506	0.5%	70.5
file_PERM	499193	66.2%	16352	16.2%	30.5
file_TEMP	219430	29.1%	84327	83.3%	2.6
owner_NET	249733	33.1%	46207	45.7%	5.4
owner_SYSTEM	392790	52.1%	25062	24.8%	15.7
owner_USER	111762	14.8%	30822	30.5%	3.6
no cut	754285	100%	101185	100%	7.5

Table 5-5: Class and owner of opened regular files

only. Log files were generally opened write-only.

Perm files are categorized by their function in Table 5-7. This categorization was done using the directories that files appeared in and/or based on file names and extensions. "System configuration" files are those appearing in / and /etc. Examples are /vmunix (the bootable kernel image) and /etc/passwd (passwords and other information on accounts). "Rwho daemon" files are used to maintain status information about machines on the network. "Library" files are those in /lib, /usr/lib and so on (these include both program libraries and additional configuration files). Files with names beginning with "." are grouped into the category "personal configuration."

cut	read-only		write-only		read/write		total
	opens	fraction	opens	fraction	opens	fraction	opens
file_LOG	735	2.1%	34819	97.7%	97	0.3%	35651
file_PERM	282853	56.7%	180976	36.3%	35200	7.1%	499029
file_TEMP	104828	47.8%	96766	44.1%	17794	8.1%	219388
owner_NET	148150	59.3%	79739	31.9%	21830	8.7%	249719
owner_SYSTEM	175787	44.8%	198183	50.5%	18712	4.8%	392682
owner_USER	64479	57.7%	34639	31.0%	12549	11.2%	111667
ruid_NET	146993	58.8%	79111	31.7%	23713	9.5%	249817
ruid_SYSTEM	99205	33.3%	188233	63.1%	10723	3.6%	298161
ruid_USER	141822	69.0%	45189	22.0%	18654	9.1%	205665
no cut	388416	51.5%	312561	41.4%	53091	7.0%	754068

Table 5-6: Mode of open for open-close sessions

These files traditionally contain startup commands and status information for various programs and are used to tailor and maintain an individual's environment. Examples include .login, .profile and .newsrc. The rest of the categories have the obvious meaning. Note that over half of the opens to perm files were made to 0.7% of the files (those in the first two categories). These files were basically all system configuration and status files. Activity to these two categories represents roughly 40% of the total file opens we observed, indicating that a substantial fraction of the system activity was devoted to communicating and maintaining information about itself and about other hosts on the network.

5.5.1.2. Per Open Results

The open activity over time is shown in Figure 5-1. Opens followed a daily pattern with a busy period between 9am and 6pm, overlaid by strong bursts due to net activity (mostly news expiration and news reception). Weekends were relatively quiet.

Figure 5-2 plots the open activity for just the first day of the trace. This shows the work day busy period more clearly. Looking closely, we can see that user activity accounted for roughly half of the daytime load. System opens had a base level (the rwho daemon) overlaid by activity that

category	opens	% opens	files	% files	opens/file
system configuration	123481	24.7%	100	0.6%	1235
rwho daemon	166761	33.4%	13	0.1%	12830
library	59245	11.9%	222	1.4%	267
manual pages	18371	3.7%	1597	9.8%	11.5
news	40022	8.0%	5828	35.6%	6.9
program source	10596	2.1%	1499	9.2%	7.1
includes	13767	2.8%	344	2.1%	40
objects	5618	1.1%	468	2.9%	12
personal configuration	23125	4.6%	1676	10.2%	13.8
mail spool	3621	0.7%	524	3.2%	6.9
other	34586	6.9%	4081	25.0%	8.5

Table 5-7: Function of opened perm files

followed or lagged slightly behind user and net activity. That is, a significant part of the system activity was indirectly due to the other classes. This activity may be attributed to logins, spoolers, mailers and so on.

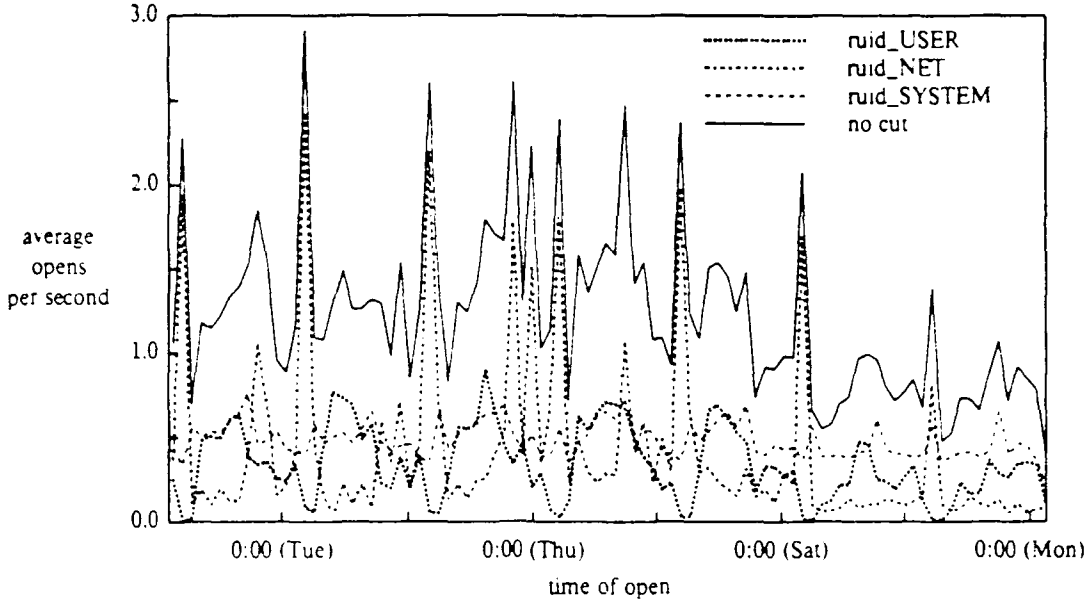


Figure 5-1: Average number of regular file opens per second (~2 hour resolution)

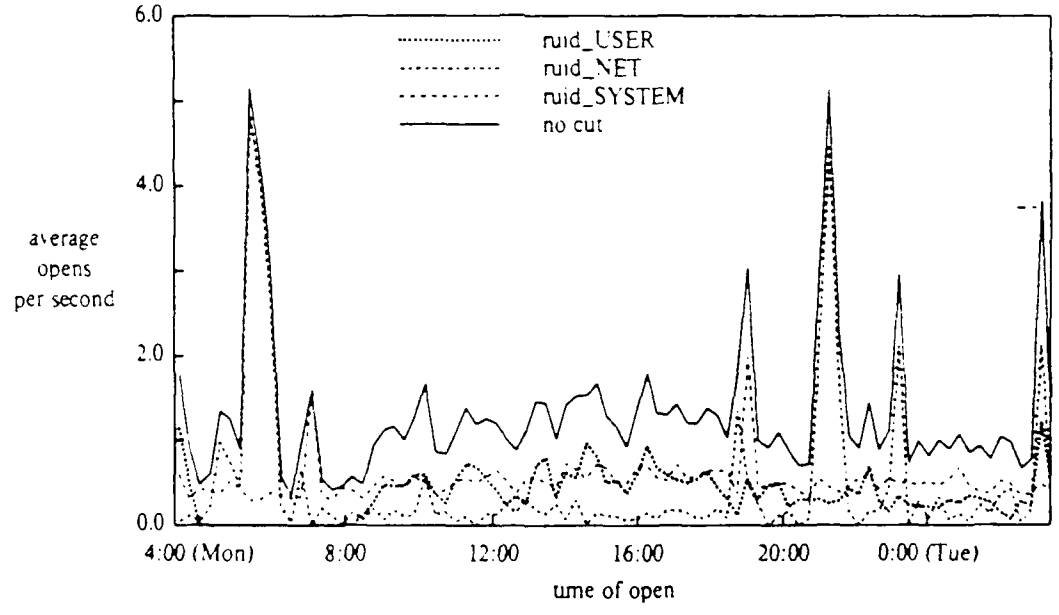


Figure 5-2: Average number of regular file opens per second (~15 minute resolution)

The read and write activity to regular files corresponded only roughly to the open activity. This can be seen by comparing Figures 5-3 and 5-4 with Figure 5-1[4]. Reads and (especially) writes were fairly bursty on the resolution used in these figures (about 2 hours). The burstiness increased as the resolution used increased. Figure 5-5 shows the throughput of the file system during a typical period of heavy user activity, averaged over 10-second intervals. This represents activity for about 25 logged-in users. It is interesting to note that the peak rates in this figure, 35K bytes/second, would present little problem for today's LAN technologies, even with fairly hefty open and transfer protocol overheads. Our results here are similar to those presented by Ousterhout et al. [Ousterhout 85] and support their contention that such networks can support large numbers of users.

Table 5-8 shows the average throughput of the file system over the life of the trace for each class of user. Note that reads accounted for 84% of the bytes transferred. Users accounted for over half of all bytes transferred, even though they made only about a quarter of the opens to regular

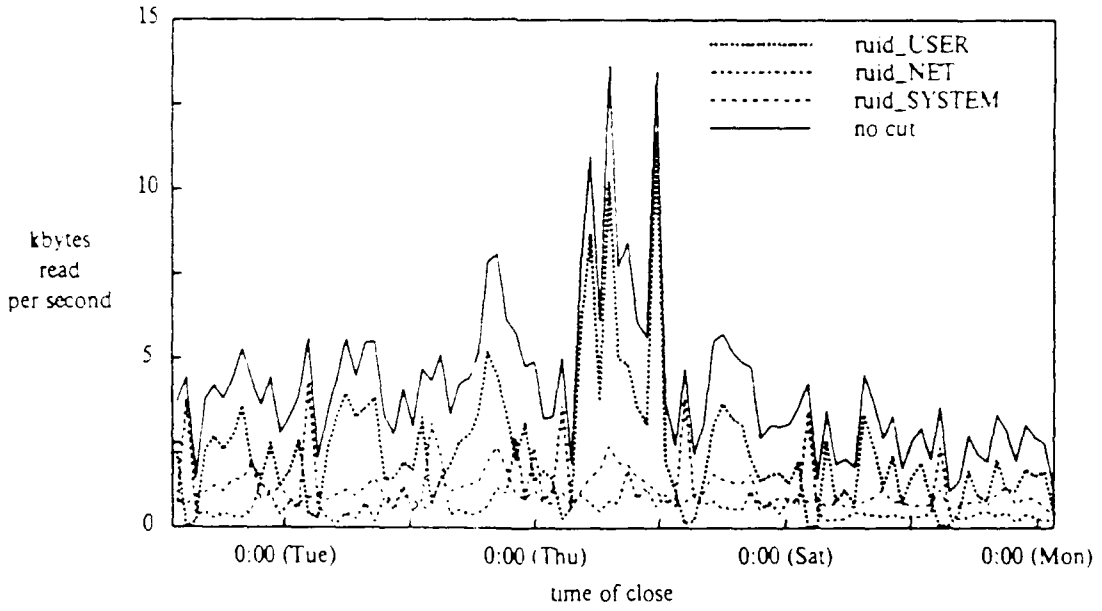


Figure 5-3: Bytes read from regular files (~2 hour resolution)

[4]The unusually heavy read/write activity on Thursday was caused by repeated execution of a large user text formatting job (formatting a Ph.D. dissertation). Most of the activity was to temp files.

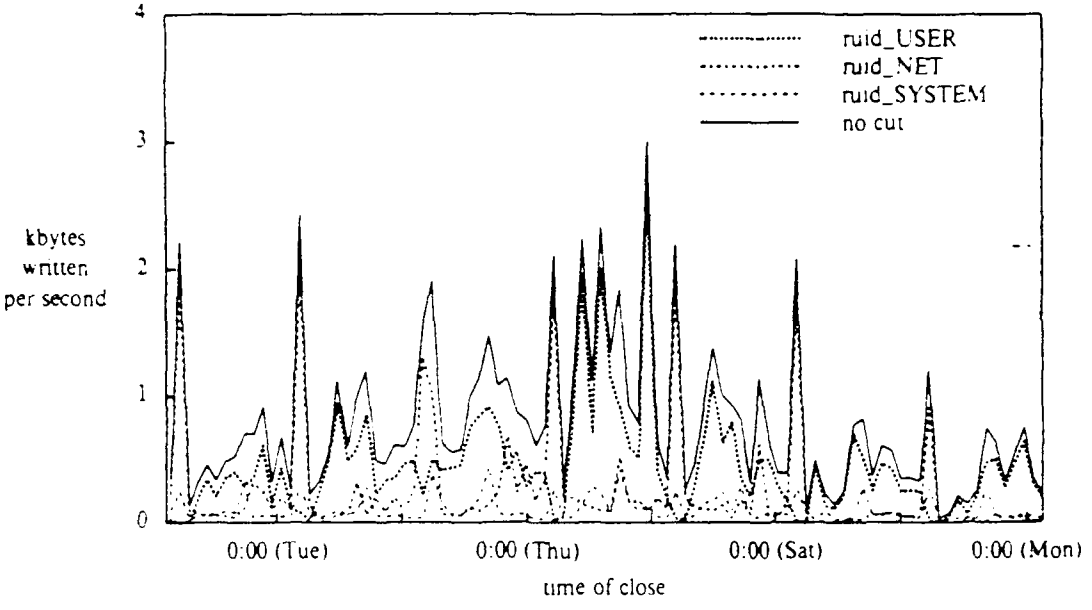


Figure 5-4: Bytes written to regular files (~2 hour resolution)

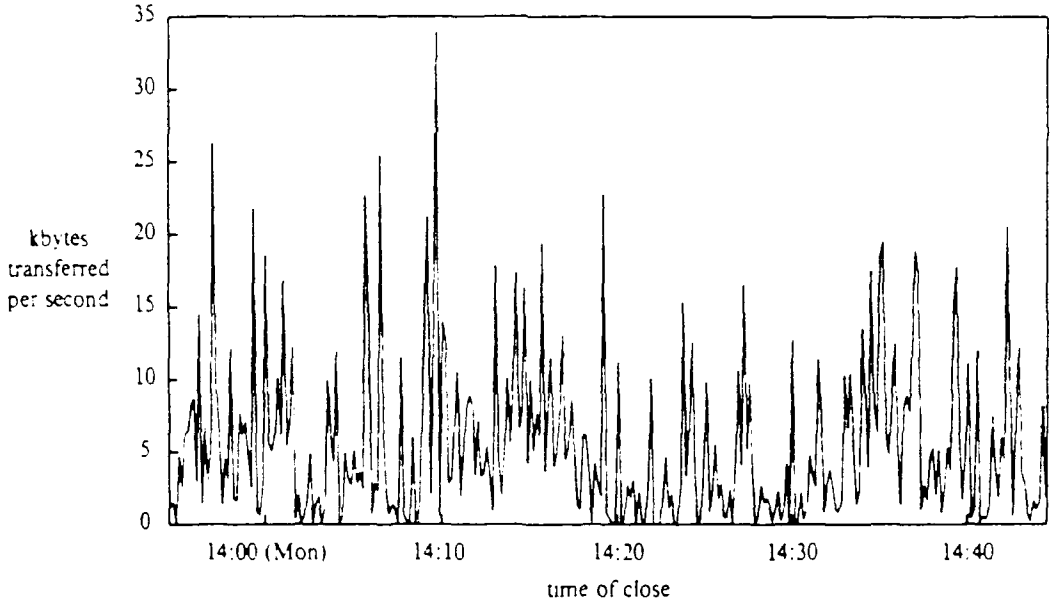


Figure 5-5: Bytes transferred to and from regular files (10 second resolution)

files (Table 5-4).

Referenced files on Seneca were small, particularly when compared to IBM mainframe environments such as the ones studied by Porcar. Figure 5-6 and Table 5-9 show file size distributions on

cut	reads		writes		overall (r+w)	
	bytes/sec	fraction	bytes/sec	fraction	bytes/sec	fraction
ruid_NET	870	21%	250	31%	1120	22.5%
ruid_SYSTEM	1060	25%	110	14%	1170	23.5%
ruid_USER	2260	54%	440	55%	2700	54%
no cut	4190	100%	800	100%	4990	100%

Table 5-8: Bytes read/written for regular files

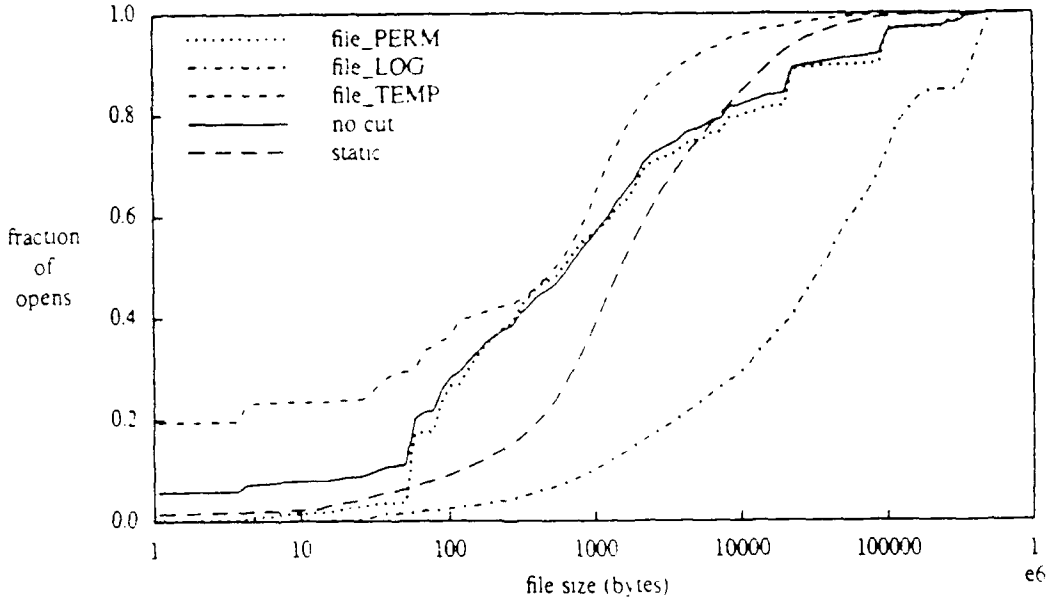


Figure 5-6: Dynamic file size distributions (cumulative, measured at close)

Seneca, weighted by the number of opens made and cut by the class of file. Note that these are *cumulative* distributions. At any point on a curve, the y value is the fraction of files with sizes less than or equal to the x value. For comparison purposes, we have included here the static file size distribution, as derived from a snapshot of the file system taken at the beginning of data collection (this is the distribution that would result if each file on the system were opened once). Table 5-9 also includes statistics for on-disk permanent files referenced during the SLAC trace.

From Figure 5-6 we see that there were substantial size differences between opened log, perm and temp files. The large number of zero length temp files was due to frequent creation of lock files

distribution	min	max	mean	median	std deviation
file_LOG, dynamic	0	1.28e6	105000	38900	1.5e5
file_PERM, dynamic	0	2.49e6	19600	620	5.9e4
file_TEMP, dynamic	0	1.3e6	2980	620	1.9e4
all, dynamic	0	2.49e6	18800	710	6.2e4
all, static	0	7.95e6	8020	1600	5.6e4
SLAC, disk file_PERM	0	94.0e6	549000	80000	2.3e6

Table 5-9: File size distributions

(these lock files serve as a very crude mutual exclusion mechanism). Log files, on the other hand, were generally an order of magnitude or more bigger than other files. The jump at 60 to 100 bytes in the perm file distribution was due to the rwho daemon, which was updating a set of status files describing machines in the network every 60 seconds. By comparing the dynamic and static distributions, we find that opens tended to favor small files (due to lock and rwho daemon files) and, to a lesser extent, a few larger files (administrative files such as /etc/passwd).

The small size of opened files (55% are under 1024 bytes, a common block transfer size, and 75% are under 4096 bytes) suggests that directory lookup and open overhead will play a large part in file access times, particularly in a distributed environment.

While most files opened in our environment were small, the majority of bytes came from files that were much larger; 2/3 of all bytes were read from files greater than 20,000 bytes long. This is shown by Figure 5-7 and Table 5-10, which give distributions for the size of opened files, weighted by the number of bytes read. We have also included here, for comparison purposes, the static space used distribution (the distribution that would result if each file on the system were completely read once). The staircase effect in the dynamic distributions is due to repeated opens and reads of a few large administrative files. /etc/passwd, for example, at 21,000 bytes, accounts for almost 20% of the bytes read. This file is infrequently modified and so would be a good candidate for replication in a distributed environment. We saw earlier (Table 5-7) that a relatively small number of files received a high fraction of the open traffic. Figure 5-7 gives graphic evidence of the corresponding impact on I/O traffic.

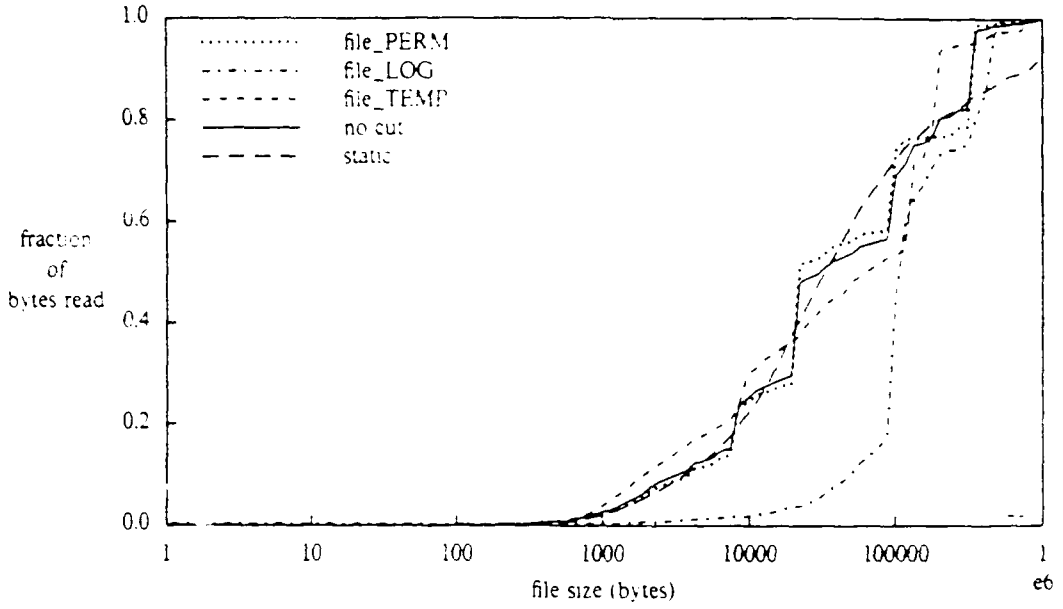


Figure 5-7: Dynamic file size distributions, weighted by bytes read (cumulative, at close)

distribution	min	max	mean	median	std deviation
file_LOG, dynamic	0	1.28e6	1.91e5	1.2e5	1.6e5
file_PERM, dynamic	0	2.49e6	1.19e5	2.2e4	2.0e5
file_TEMP, dynamic	0	1.3e6	1.12e5	6.8e4	1.5e5
all, dynamic	0	2.49e6	1.19e5	3.4e4	1.9e5
all, static	0	7.95e6	3.6e4	1.29e4	5.1e4

Table 5-10: File sizes, weighted by number of bytes read

Our distributions for the overall sizes of opened files and for the source of bytes read (Figures 5-6 and 5-7) agree with the distributions found by Ousterhout et al.. By these measures, at least, our data appear to be representative of a university research environment.

Two figures that are useful in estimating the appropriateness of dynamic migration are the fraction of a file opened for reading that is actually read and the fraction of a file opened for writing that is actually written. As mentioned in section 5.3, we don't have complete information on which bytes of opened files were read and written. However, if we make the reasonable (for our environment) assumption that a given byte in a file was not usually read or written repeatedly in a single session, we can use the counts of bytes read and written from the close record to calculate the

fraction of a file read or written. Figure 5-8 shows the percentage read for files opened read-only, cut by the class of the file. Figure 5-9 shows the percentage written for files opened write-only and Figures 5-10 and 5-11 are for files opened read/write. In all cases, the size used is the size of the file when closed. Zero length files are omitted. Tables 5-11 through 5-14 provide some statistics on the distributions in these figures.

From these figures we see that most opens with read-only or write-only access resulted in the file being completely read or written. The notable exception was for log files. For these files, writes usually just incrementally extended the file. This is shown clearly in Figure 5-9 and indicates that we have successfully extracted log files from our data. Much less can be said about the read/write behavior of files opened with read/write access. For these files, information on usage history or more detailed information on the intended usage of the file would be needed to predict the read/write behavior. Recall (Table 5-6) that this category represents only 7% of the opens and so the additional information will not usually be needed.

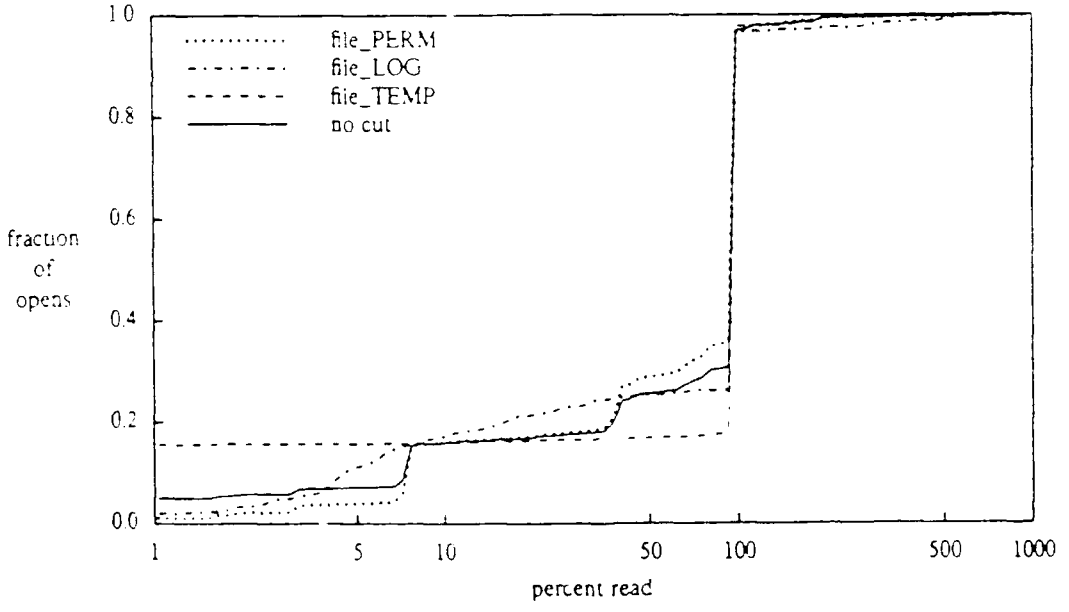


Figure 5-8: Percent of file read for read-only opens (cumulative)

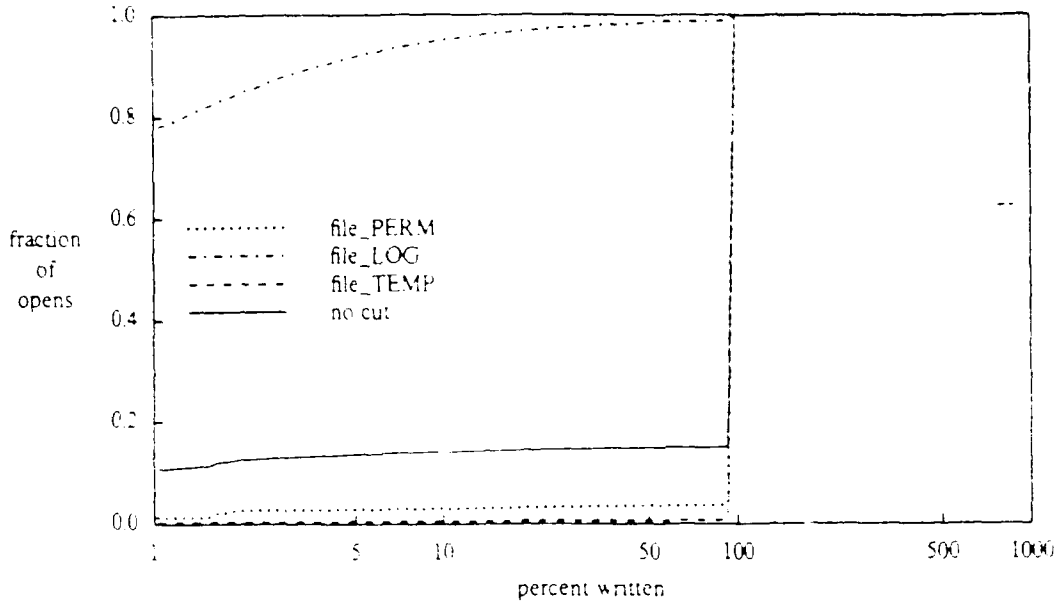


Figure 5-9: Percent of file written for write-only opens (cumulative)

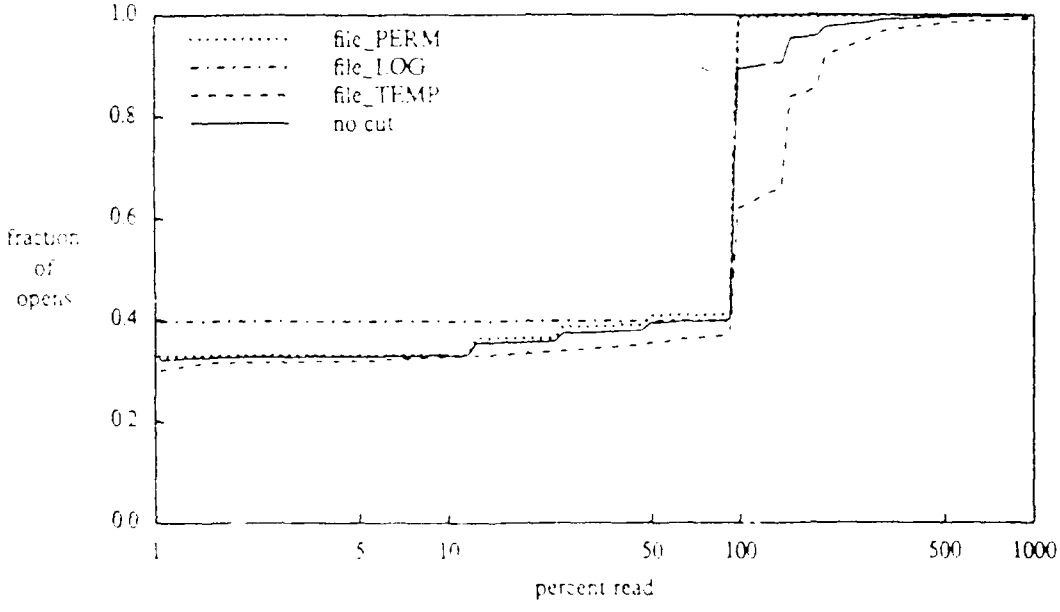


Figure 5-10: Percent of file read for read/write opens (cumulative)

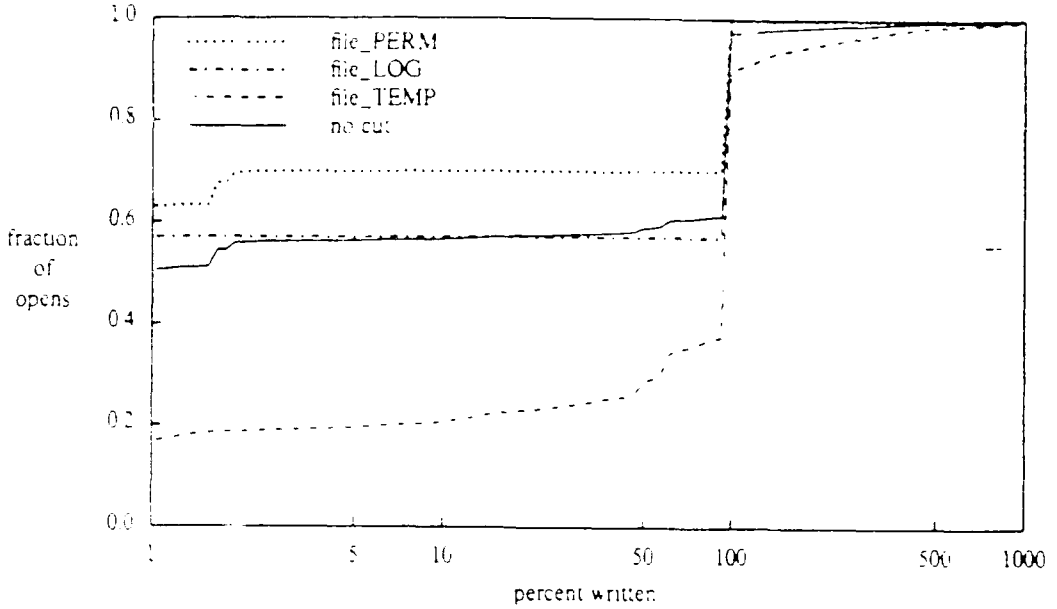


Figure 5-11: Percent of file written for read/write opens (cumulative)

distribution	min	max	mean	median	std dev	<100%	>100%
file_LOG	0	690	85.8	100	72	26%	3.3%
file_PERM	0	64100	83.2	100	235	36%	3.2%
file_TEMP	0	3600	85.8	100	53	18%	2.1%
no cut	0	64100	83.9	100	202	31%	2.9%

Table 5-11: Percentage read (read-only opens)

distribution	min	max	mean	median	std dev	<100%	>100%
file_LOG	0	100	2.8	<1	12	98.8%	0%
file_PERM	0	200	96.6	100	18	3.9%	0%
file_TEMP	0	9600	100.8	100	85	0.7%	0.2%
no cut	0	9600	85.7	100	53	15%	0.1%

Table 5-12: Percentage written (write-only opens)

distribution	min	max	mean	median	std dev	<100%	>100%
file_LOG	0	100	60.2	100	49	40%	0%
file_PERM	0	1900	62.5	100	66	41%	0.4%
file_TEMP	0	65000	138	100	1180	37%	37%
no cut	0	65000	82.9	100	615	40%	11%

Table 5-13: Percentage read (read/write opens)

distribution	min	max	mean	median	std dev	<100%	>100%
file_LOG	0	100	43.0	<1	49.5	57%	0%
file_PERM	0	20000	36.4	<1	275	70%	0.1%
file_TEMP	0	3600	93.5	100	150	37%	9.8%
no cut	0	20000	51.8	<1	249	61%	2.7%

Table 5-14: Percentage written (read/write opens)

Overall, 68% of files opened with read access (read-only or read/write) were completely read and 78% of files opened with write access (write-only or read/write) were completely written. This may be contrasted with the SLAC data, where only 17% of opened permanent files were completely accessed. The high percentage of files completely accessed on Seneca is due to the much smaller file size and to the lack of any serious database activity.

As one might expect, the fraction of a file that was accessed depended strongly on the size of the file. Very small files were usually completely read or written. Large files were rarely completely read or written. This is shown for files opened read-only and write-only in Figures 5-12 and 5-13 and in Tables 5-15 and 5-16. Files opened with read/write access followed a similar pattern.

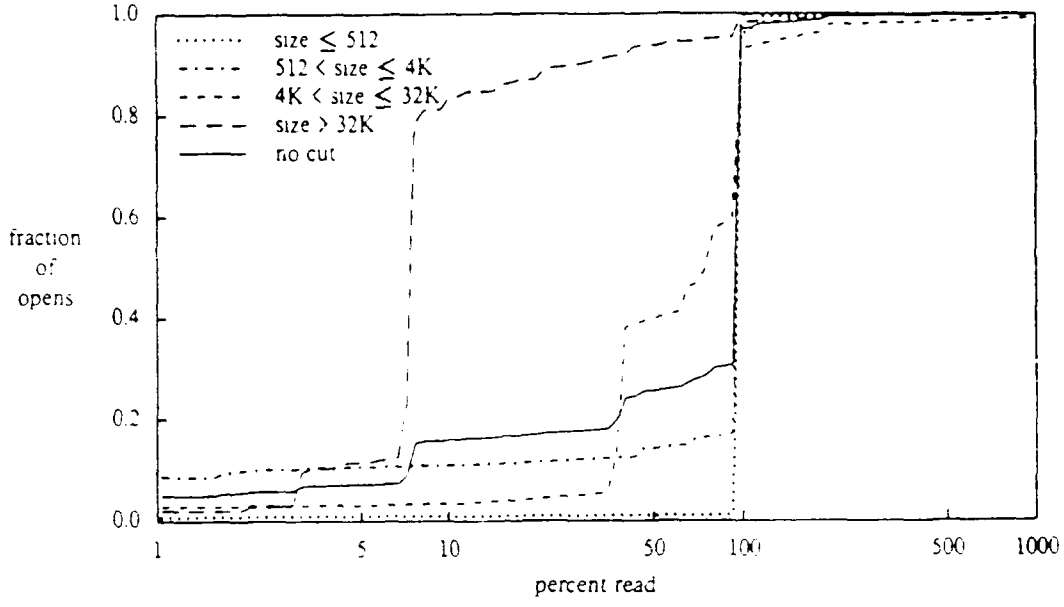


Figure 5-12: Percent of file read for read-only opens (cumulative, by size)

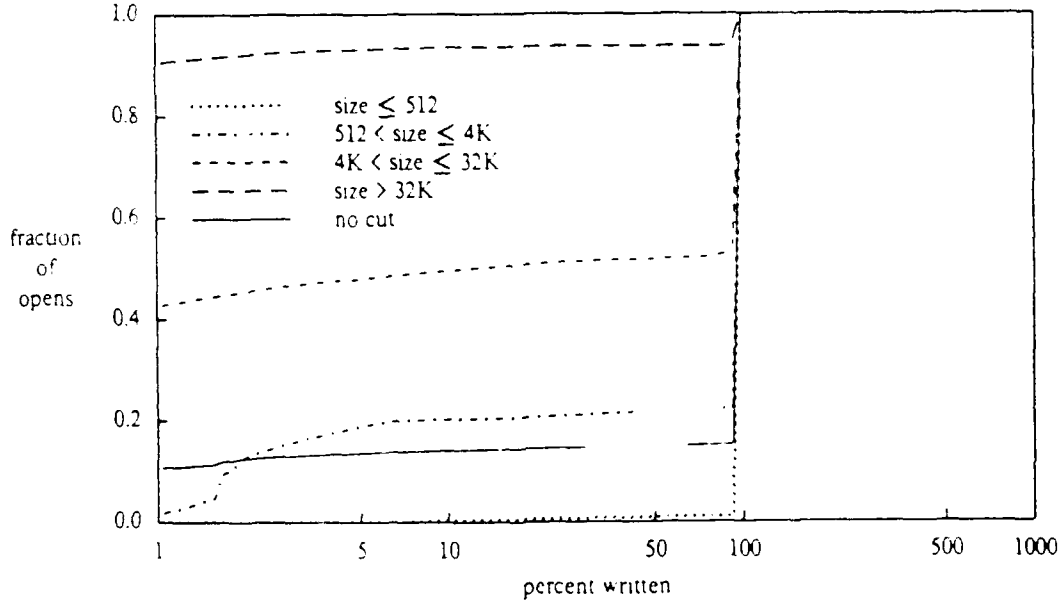


Figure 5-13: Percent of file written for write-only opens (cumulative, by size)

size	fraction of r-o opens	min	max	mean	median	std dev	<100%	>100%
≤ 512 bytes	22.7%	0	3600	100.1	100	19	1.2%	3.8%
512 < size ≤ 4K	46.7%	0	64100	88.4	100	158	17%	1.7%
4K < size ≤ 32K	18.5%	0	55500	97.0	80	382	59%	6.7%
size > 32K bytes	12.1%	0	12500	15.6	7.7	110	95%	0.4%
all	100%	0	64100	83.9	100	202	31%	2.9%

Table 5-15: Percentage read, by size (read-only opens)

size	fraction of w-o opens	min	max	mean	median	std dev	<100%	>100%
≤ 512 bytes	71.5%	0	6800	99.5	100	35	1.0%	0.1%
512 < size ≤ 4K	13.0%	0	9600	79.7	100	84	22%	0.3%
4K < size ≤ 32K	7.3%	0	101	48.8	12	49	54%	0%
size > 32K bytes	8.2%	0	100	6.5	<1	24	94.2%	0%
all	100%	0	9600	85.7	100	53	15%	0.1%

Table 5-16: Percentage written, by size (write-only opens)

5.5.1.3. Per File Results

The number of opens per file gives an indication of the potential benefits and penalties of migrating files to a user's machine (the degree of sharing is also a factor here). Most files in our environment were opened only once or twice (Figure 5-14 and Table 5-17). This may be attributed to the large number of lightly used temp files; log and perm files saw considerably more activity. We have also included in Table 5-17 information on the distribution for on-disk permanent files in the SLAC trace (for a period of 310 hours). SLAC perm files saw, on average, considerably less activity than the perm files in our environment, despite the longer SLAC logging period.

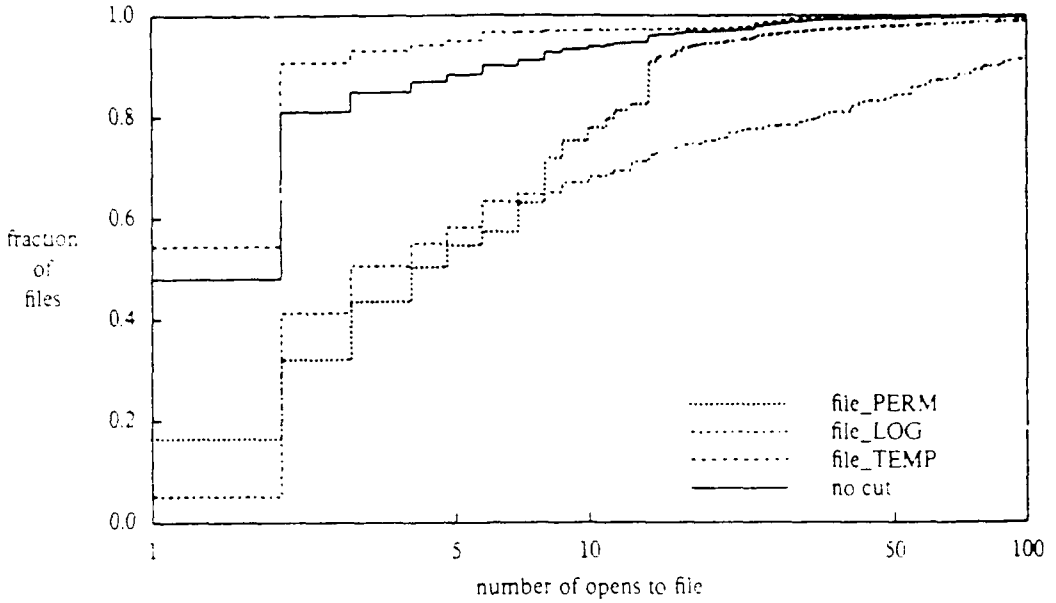


Figure 5-14: Number of opens per active file (cumulative)

distribution	mean	median	opened once	opened twice	opened more than twice	max
file_LOG	70.5	3	5%	36%	59%	5330
file_PERM	30.5	4	16%	16%	68%	26800
file_TEMP	2.6	1	55%	36%	9%	1920
no cut	7.5	2	48%	33%	19%	26800
SLAC, disk file_PERM	12	2	46%	23%	31%	2660

Table 5-17: Number of opens/file

distribution	mean	median	std dev
file_LOG	1480	950	1700
file_PERM	8120	>5000	8100
file_TEMP	57	3	210
no cut	5400	480	7600

Table 5-18: Open distribution (as a function of opens/file)

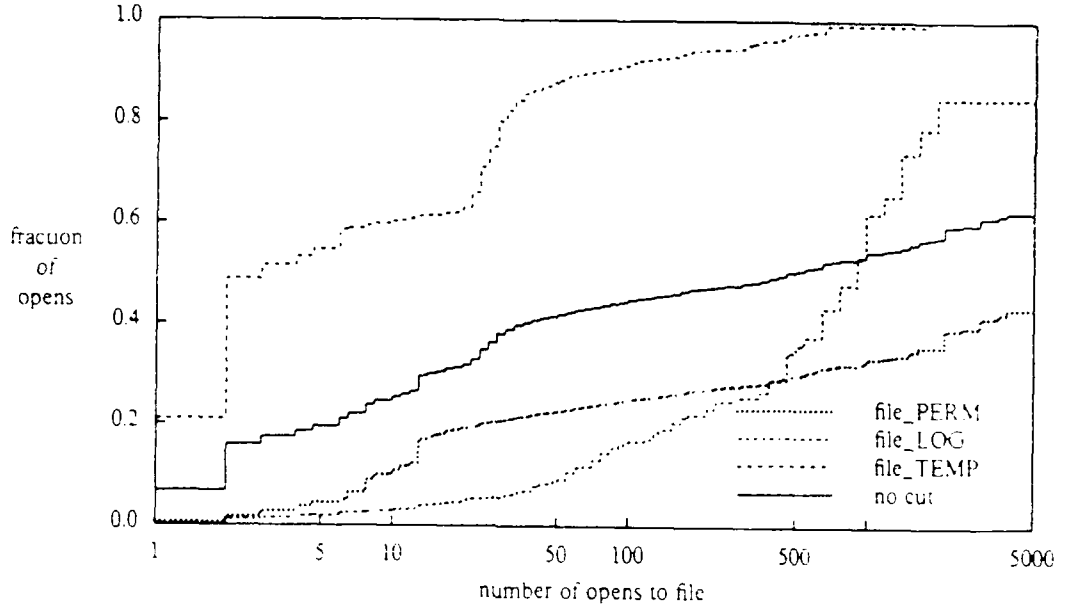


Figure 5-15: Fraction of opens per active file (cumulative)

Most opens went to files opened many times. 75% went to files opened more than 10 times and half to files opened more than 480 times (Figure 5-15 and Table 5-18). The most frequently opened files on Seneca (Table 5-19[5]) were administrative and configuration files in /etc and in library directories, rwho daemon files and news databases.

Files in our environment were usually only open for a few tenths of a second (Figure 5-16 and Table 5-20). Temp files were open for relatively long periods of time. This is to be expected, since they are often used to store intermediate results as they are being calculated. The distribution for perm files is consistent with the small files sizes and whole file transfers we saw earlier. Programs open these files, transfer data and then immediately close the files.

[5]This table actually lists the most frequently accessed Inodes, with the given name being the path used to first access the inode. For the most part, this distinction doesn't matter. There are a few inodes listed here, though, that are one of several versions of a heavily used system file. An example is /etc/passwd, which starts life as /etc/pump. Occurrences of this are noted in the table.

rank	opens	fraction	path of first open
1	26801	5.4%	/etc/hosts
2	20675	4.1%	/usr/spool/rwho/whod.keuka
...	...	...	[2 more rwho daemon files]
5	14977	3.0%	/etc/passwd [35485 (7.1%) with /etc/ptmp versions]
6	12036	2.4%	/etc/utmp
7	10594	2.1%	/usr/spool/rwho/whod.capella
...	...	...	[9 more rwho daemon files]
17	9386	1.9%	/usr/include/whoami.h
18	8881	1.8%	/etc/ptmp [version of /etc/passwd]
19	8630	1.7%	/etc/ptmp [version of /etc/passwd]
20	7533	1.5%	/usr/lib/sendmail.st
21	7295	1.5%	/vmunix
22	6908	1.4%	/etc/termcap
23	6400	1.3%	/etc/group [6947 (1.4%) for all versions]
24	6294	1.3%	/etc/services
25	6211	1.2%	/etc/gettytab

Table 5-19: Frequently opened inodes

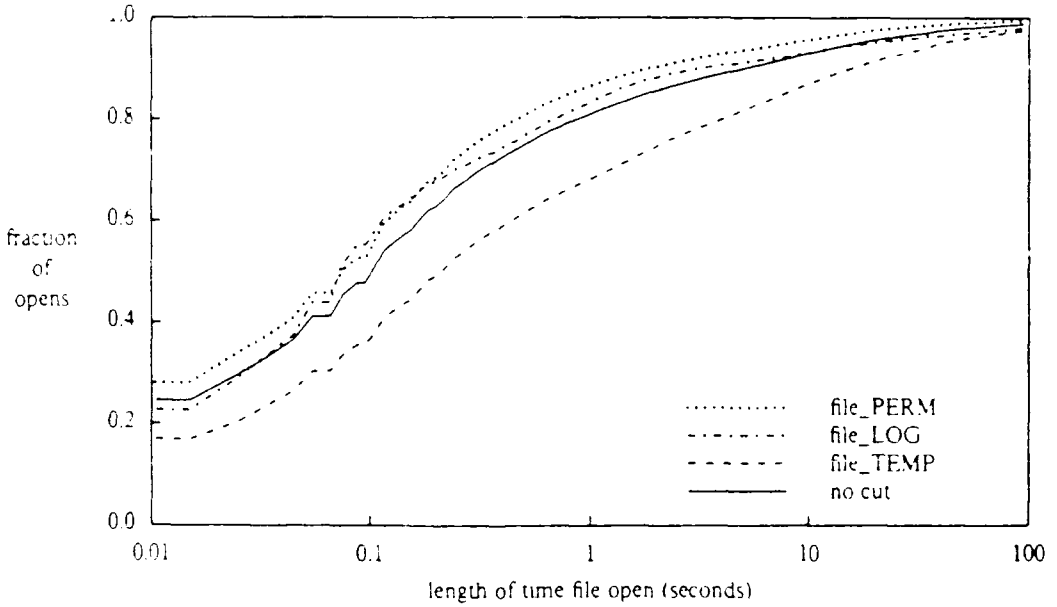


Figure 5-16: Times from file open to close (cumulative)

Knowledge of file interopen intervals (the time from one open of a file to the next) is useful in estimating both the appropriate time scale for migration and the possibilities for caching. Figure 5-17 and Table 5-21 show that interopen intervals in our environment were short (opens to a file

distribution	min	max	mean	median	std deviation
file_LOG	0	8.6e4	33.4	0.08	1140
file_PERM	0	7.6e4	6.5	0.08	251
file_TEMP	0	4.8e4	20.5	0.22	335
no cut	0	8.6e4	11.8	0.1	369

Table 5-20: Open time (seconds)

distribution	min	max	mean	median	std deviation
file_LOG	0	5.4e5	965	15	9.8e3
file_PERM	0	5.4e5	8215	60	2.4e4
file_TEMP	0	4.2e5	6655	3.6	1.8e4
no cut	0	5.4e5	7502	60	2.2e4
SLAC, disk file_PERM	0	9.7e4	8350	50	4.4e4

Table 5-21: File interopen intervals (seconds)

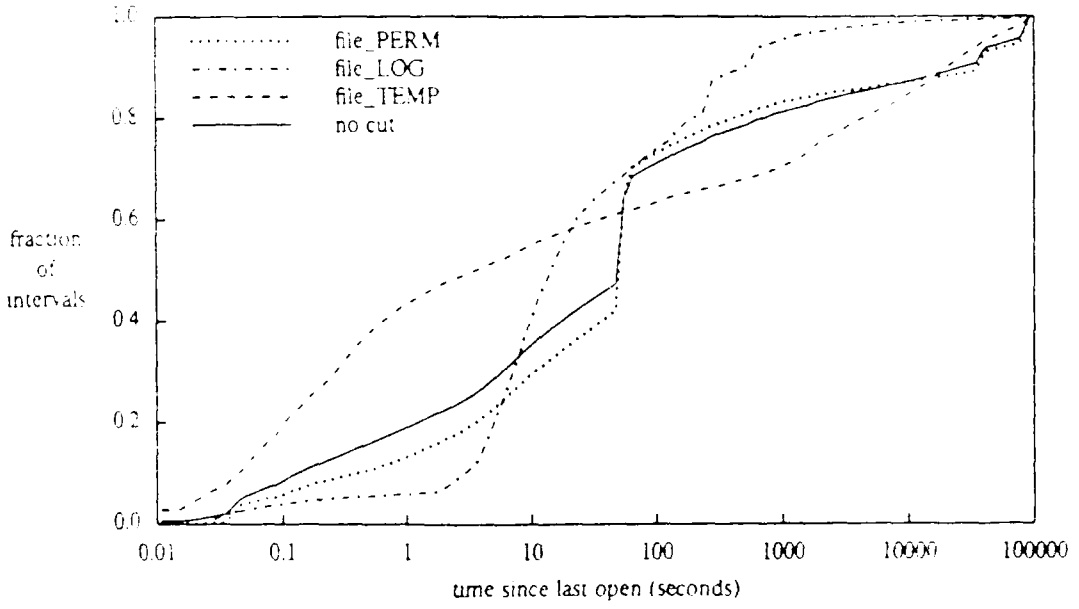


Figure 5-17: File interopen intervals (cumulative)

were strongly clustered). When a file was opened, the following open (if any) had a 50% probability of occurring within the next 60 seconds. Interopen intervals for temp files were particularly short. If a temp file was opened multiple times (many were not), the next open often occurred within a few seconds of the last one. This is to be expected for files that are used to hold results between job steps. Log files also had shorter interopen intervals than files as a whole. Most log file opens were made by net processes and these processes show intense bursts of activity (Figure 5-2), so this is not surprising. The jump at 60 seconds in the distribution for perm files is due to rwho daemon activity.

The lifetime of a file in our environment depended strongly on the class of the file. Most temp files lived less than a minute. The overwhelming majority of perm files had lifetimes that extended beyond the logging period. Log files fell in between (mostly due to short lived UUCP work logs). File lifetime distributions are shown in Figure 5-18. Here files that existed before logging was started or that continued to exist after logging was terminated were given lifetimes exceeding the logging period (lie to the right of the histogram). Because so many log and perm files fell into this category, we have not included the moments of these distributions.

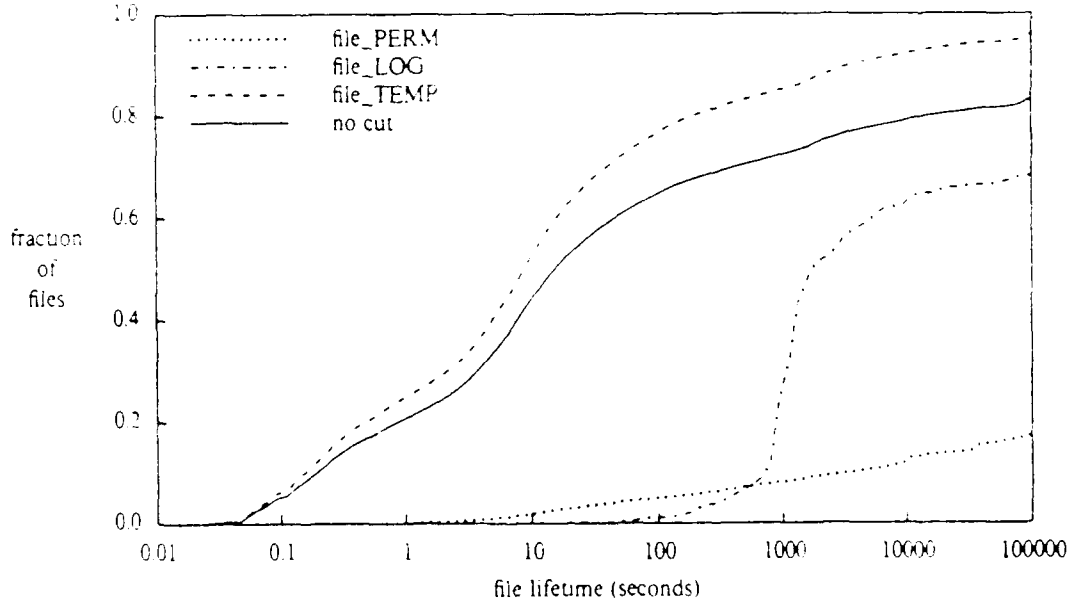


Figure 5-18: File lifetimes (cumulative)

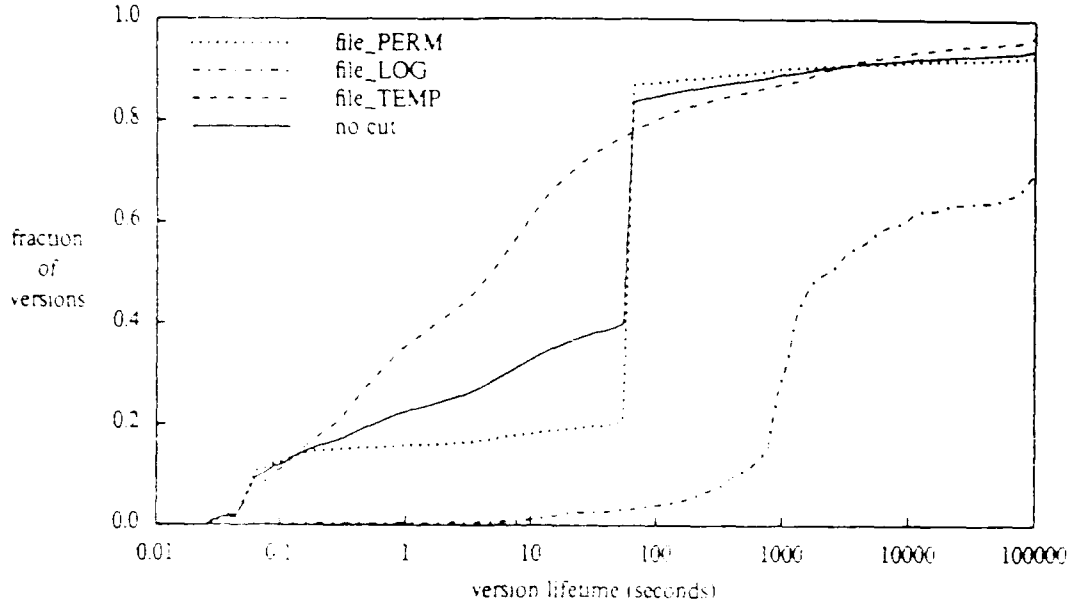


Figure 5-19: Version lifetimes (cumulative)

Even though most perm files have long happy lifetimes, the data in these files is not so fortunate. This is shown in Figure 5-19, where we have histogrammed the time from when a file is created or written to the time when a file is overwritten or deleted (this is the file lifetime used by Ousterhout et al.). Files that were only partially written are not included in this histogram. Again, data whose lifetime extended beyond the limits of our log were given lifetimes exceeding the logging period. The large jump at 60 seconds is due to rwho daemon activity. Since we include all files here and Ousterhout et al. included just new data, our results are not directly comparable.

The first two columns of Table 5-22 show the mean number of readers per file, as indicated by the account (ruid) of the reader, and the percentage of files with more than one reader, cut by the file class and owner. The next four columns show this information for writers and for the overall number of file users. The last two columns show the mean and maximum number of *inversions* per file. The number of inversions is the number of times that the most recent user of the file changes (this is basically the inversion clustering metric used by Porcar [Porcar 82]). For a file used by only one user, the number of inversions will be zero.

We can see from Table 5-22 that 11.8% of the files seen duri..g the logging period were accessed by multiple users (users with separate accounts). Multiple readers were much more common than multiple writers. Most shared files belonged to net. These were predominately news articles (perm files). Logs were also heavily shared. They frequently had multiple writers and separate readers. Although system files were not as heavily shared as net files, in terms of the number of shared files, the high mean number of inversions (2.92) indicates that the system files that were shared were not shy about it. Few user files were shared, and the low mean number of inversions (0.111) indicates that this sharing was incidental to the normal use of user files.

The overall distributions are shown in more detail in Table 5-23. Note that very few files had more than 2 writers and that even the distribution of the number of users per file drops off quite sharply.

5.5.2. Execute Patterns

The basic calls to run an executable file under 4.2BSD UNIX are execv and execve. These calls are grouped together under the heading "execute" in Table 5-3. Users were responsible for half of the execute requests in our log (Table 5-23), even though, as we saw in section 5.5.1, they made only a quarter of the opens to regular files. Most executes were done on system files. Users

cut	readers		writers		users (r / w)		inversions	
	mean	1	mean	>1	mean	>1	mean	max
file_LOG	0.98	3.8%	1.86	26.7%	2.67	76.9%	5.25	293
file_PERM	1.575	17.9%	0.444	3.4%	1.711	20.9%	5.52	12529
file_TEMP	0.639	4.4%	1.02	2.1%	1.212	9.6%	0.293	92
owner_NET	0.905	11.9%	0.933	4.1%	1.501	20.7%	0.874	1288
owner_SYSTEM	0.483	3.0%	0.962	0.12%	1.196	3.8%	2.92	12529
owner_USER	0.85	1.3%	0.876	0.84%	1.053	2.7%	0.111	169
no cut	0.792	6.6%	0.930	2.4%	1.30	11.8%	1.16	12529

Table 5-22: file sharing, by file class and owner

number	readers		writers		users (r + w)		inversions	
	count	cum	count	cum	count	cum	count	cum
0	44711	44.2%	11252	11.1%	-	-	89272	88.2%
1	49845	93.4%	87510	97.6%	89272	88.2%	5838	94.0%
2	3022	96.4%	2009	99.59%	7555	95.7%	2182	96.2%
3	1244	97.7%	209	99.80%	1818	97.5%	666	96.8%
4	685	98.3%	66	99.86%	758	98.2%	799	97.6%
5	421	98.8%	23	99.89%	448	98.7%	389	98.0%
6	356	99.11%	18	99.90%	369	99.05%	339	98.3%
7	245	99.35%	20	99.92%	258	99.30%	318	98.6%
8	167	99.52%	8	99.93%	175	99.47%	291	98.9%
9	80	99.60%	12	99.94%	88	99.56%	213	99.13%
10	105	99.70%	8	99.95%	111	99.67%	156	99.29%
>10	304	100%	50	100%	333	100%	722	100%
total	101185	-	101185	-	101185	-	101185	-

Table 5-23: readers, writers, users and inversions; no cuts

owned almost half of the executables seen but there were few executes of these files.

Most executable files were between 5,000 and 100,000 bytes long (Figure 5-20 and Table 5-25). The relatively large size of executables is a reflection of the lack of run-time library sharing. All executables contain whatever code they need to run.

cut	executes	% executes	executables	% executables	executes/executable
ruid_NET	26761	21.4%	41	7.1%	653
ruid_SYSTEM	38093	30.5%	137	23.6%	278
ruid_USER	60210	48.1%	528	90.9%	114
owner_NET	12190	9.7%	34	5.9%	359
owner_SYSTEM	108646	86.9%	291	50.1%	373
owner_USER	4228	3.4%	256	44.1%	17
no cut	125064	100%	581	100%	215

Table 5-24: Basic active executable statistics

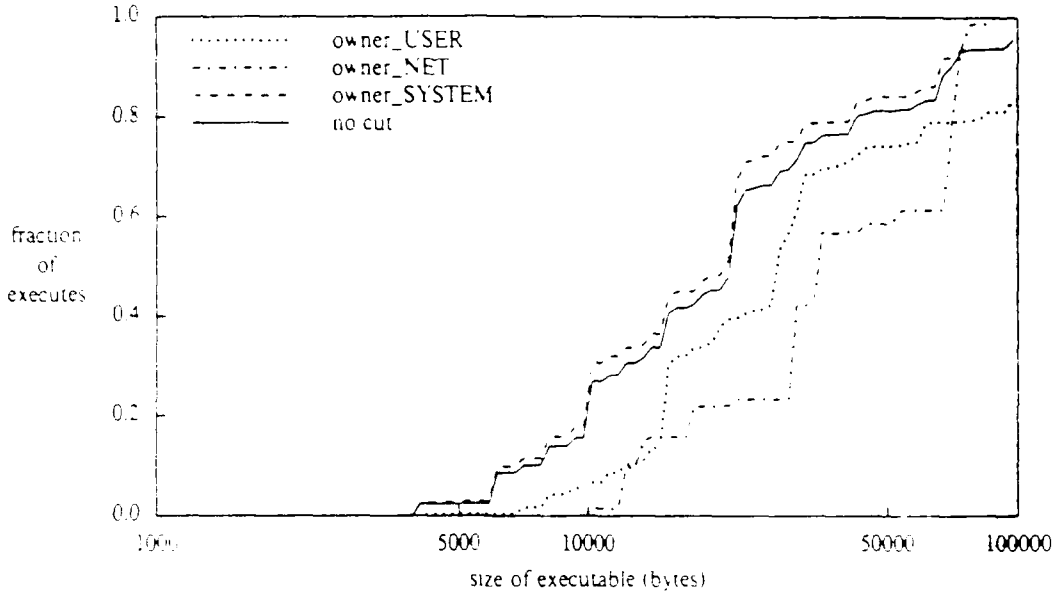


Figure 5-20: Dynamic executable file size distributions (cumulative)

distribution	min	max	mean	median	std deviation
owner_NET	9216	8.8e4	44400	35500	23900
owner_SYSTEM	4096	1.1e6	34500	21400	84900
owner_USER	4228	3.2e6	55900	28200	135000
no cut	4096	3.2e6	36200	22400	83400

Table 5-25: Executable file sizes (bytes)

distribution	min	max	mean	median	std deviation
owner_NET	1	2152	359	60	570
owner_SYSTEM	1	17519	373	24	1340
owner_USER	1	675	17	3	59
no cut	1	17519	215	8	970

Table 5-26: Number of executes/active executable

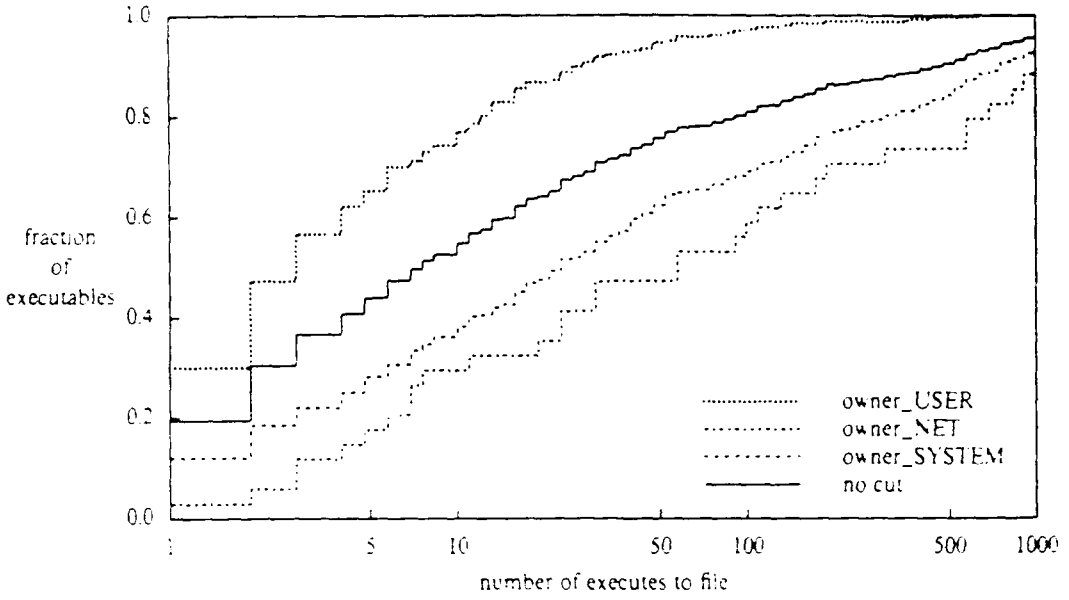


Figure 5-21: Number of executes per active executable (cumulative)

An executable file saw considerably more activity than other regular files (Figure 5-21 and Table 5-26). Almost half were executed 10 times or more. This is not surprising, considering the small number of active executables.

Most executes went to files executed a large number of times. Half went to files executed more than 2000 times (Figure 5-22 and Table 5-27) and 95% went to files executed at least 100 times.

distribution	mean	median	std dev
owner_NET	1270	1380	634
owner_SYSTEM	5150	2600	5850
owner_USER	230	105	244
no cut	4600	2000	5640

Table 5-27: execute distribution (as a function of executes/executable)

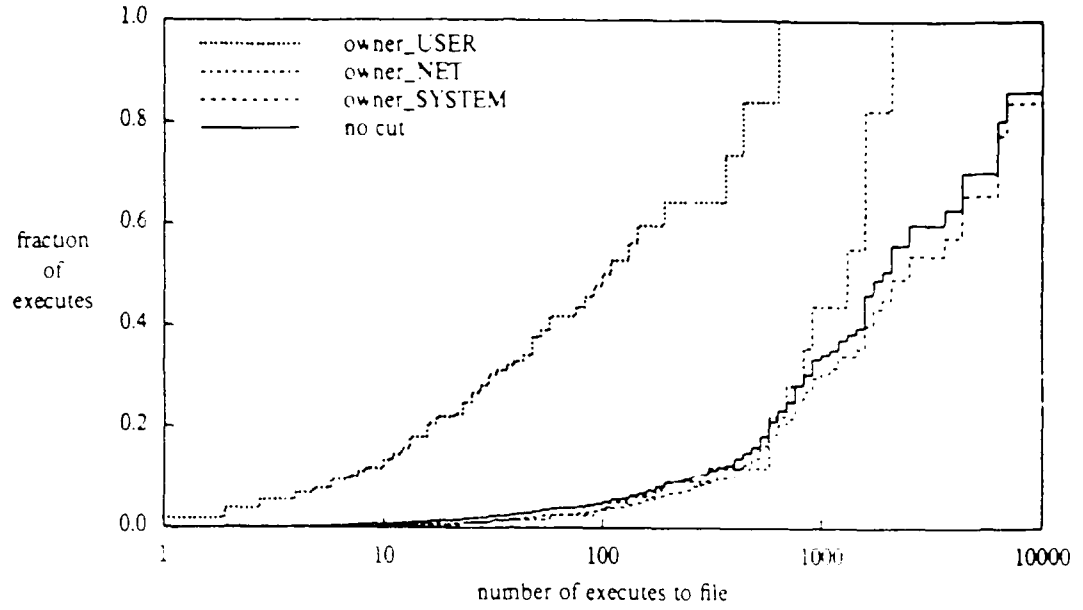


Figure 5-22: Fraction of executes per active executable (cumulative)

The most frequently executed files on Seneca were shells and system utilities to delete files, evaluate conditionals, list directories and distribute files to other machines (Table 5-28). Over half of the executes went to only 13 files. These files, taken together, occupied 0.46MB of disk space (0.08% of the total). This suggests that even a very modest amount of caching or other special treatment for frequently requested programs will produce significant improvements. Evidence for this was also seen in a study of 2MB diskless Sun workstations running a version of UNIX similar to the one on Seneca at the University of Washington [Lazowska 84]. For the Suns studied, 80% of the bytes transferred were due to file accesses and only 20% were for paging. If we take our average executable size times the execute rate (most 4.2BSD executables are loaded using demand paging), we get a very crude paging estimate of 7500 bytes/second, or about 170% of the transfers due to opens (Table 5-8). The difference between our crude estimate and the behavior seen at the University of Washington is probably due to both the caching of pages of frequently executed files and to code and debugging information in executables that is not used.

The distribution of time between executes for executables is given in Figure 5-23 and Table 5-29. These distributions lend support to our caching arguments.

rank	executes	fraction	path of first execute
1	17519	14.0%	/bin/sh
2	6946	5.6%	/bin/rm
3	6511	5.2%	/bin/[
4	6402	5.1%	/bin/csh
5	4568	3.7%	/bin/ls
6	4497	3.6%	/etc/rdist
7	3776	3.0%	/usr/ucb/more
8	2517	2.0%	/usr/ucb/vi
9	2514	2.0%	/bin/login
10	2197	1.8%	/bin/echo
11	2152	1.7%	/usr/bin/rnews
12	2143	1.7%	/usr/lib/sendmail
13	2026	1.6%	/etc/logid
14	1891	1.5%	/bin/hostname
15	1803	1.4%	/bin/rmdir

Table 5-28: Frequently executed inodes

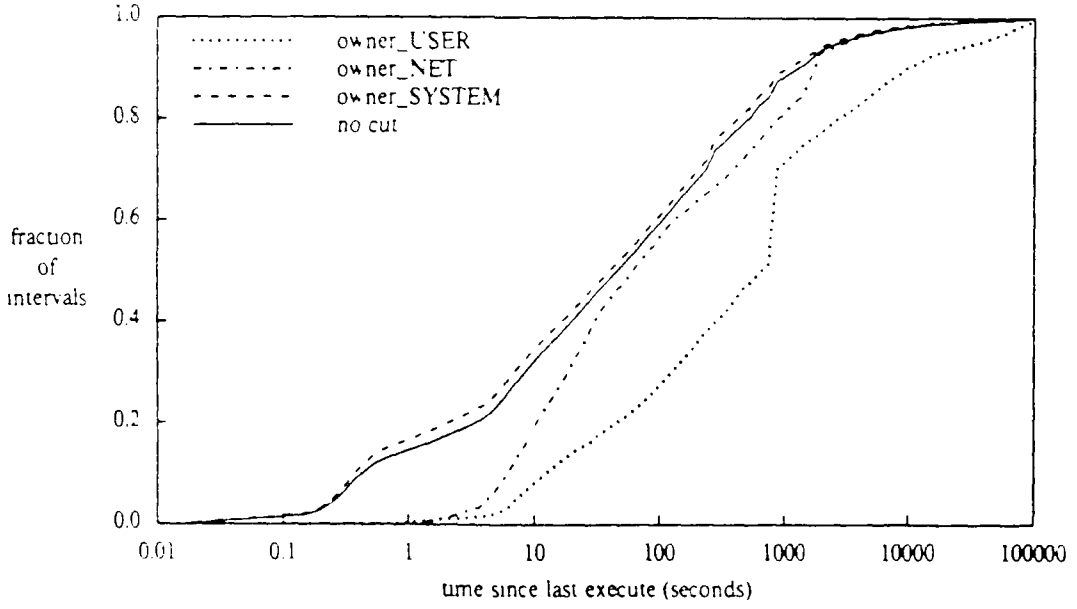


Figure 5-23: File interexecute intervals (cumulative)

Executing a program on UNIX is usually done using the sequence fork (to create a copy of the running process); execv or execve (to replace that copy with the new program); exit (when done). Since over 2/3 of the forks on Seneca were followed by an execute we can, by looking at process lifetimes (time from fork to exit) estimate how long executables were in use. Process lifetime distributions, cut by the ruid of the requester, are given in Figure 5-24 and Table 5-30[6]. Over half of all processes recorded in the log lived less than a second. System processes were particularly short-lived. With the exception of the large number of system processes that lived less than a tenth of a second (due mostly to local network servers), our results agree with process lifetime results given by Zhou et al. [Zhou 85].

Executable files were more heavily shared than opened files (Table 5-31). This should come as no surprise, since there were relatively few executables and these were usually located in public directories. User executables were relatively lightly shared.

distribution	min	max	mean	median	std deviation
owner_NET	0.05	2.2e5	1160	65	6550
owner_SYSTEM	0	5.6e5	983	40	7680
owner_USER	0.52	4.1e5	6290	730	23600
no cut	0	5.6e5	1170	47	8610

Table 5-29: Interexecute intervals (seconds)

distribution	min	max	mean	median	std deviation
ruid_NET	0.02	7250	15.7	3.0	88
ruid_SYSTEM	0.01	215000	52.2	0.09	1380
ruid_USER	0.02	76100	118	2.4	1190
no cut	0.01	215000	165	0.95	2560

Table 5-30: Process lifetimes (seconds)

[6]Some processes, such as login shells, start life in one ruid class and exit in another. These are included only in the overall distribution

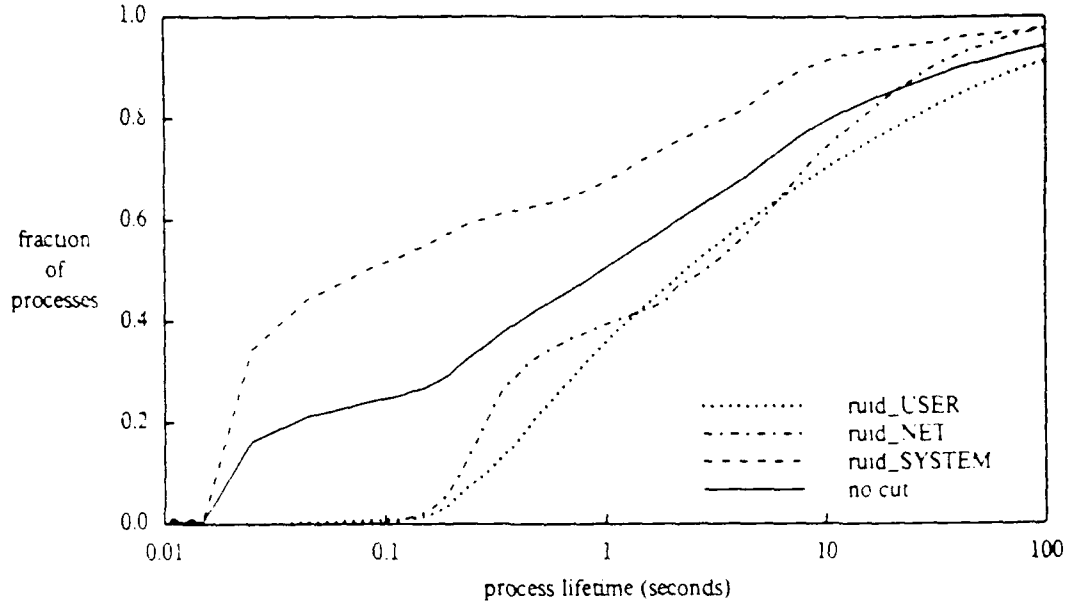


Figure 5-24: Process lifetimes (cumulative)

cut	executors					inversions			
	mean	median	>1	>5	max	mean	>0	>5	max
owner_NET	10.3	4	70.6%	41.2%	45	92.9	70.6%	55.9%	957
owner_SYSTEM	12.0	2	61.5%	34.4%	111	126	61.5%	38.8%	6539
owner_USER	1.38	1	9.8%	1.6%	28	1.74	9.8%	4.7%	205
no cut	7.2	1	39.2%	20.3%	111	69.5	39.2%	24.6%	6539

Table 5-31: Executable sharing

5.5.3. User File Patterns

In this section we take a closer look at user files. Some distributed file systems (for example, the ITC DFS [Satyanarayanan 85]) deal primarily or wholly with user files. In addition, we expect that user file access patterns will be less dependent on the operating system used. These factors make user file reference patterns particularly interesting.

The results presented in this section are actually for user references to user files (owner_USER+ruid_USER cut, referred to as the "U" cut below). These references represented

over 90% of the references to user files. The remaining references were mostly infrequent periodic references made by system processes and had little effect on the distributions we see (with the exception of some of the sharing results). The organization of this section follows closely that of section 5.5.1.

5.5.3.1. Basic Statistics for User Files

The majority (62%) of user references to user files were to perm files (Table 5-32), even though less than a third of the referenced user files were perm files. There were few references to log files. Most of these files were logs of mail sent or read and so the low level of activity is not surprising. With the exception of a somewhat higher proportion of perm files, these figures agree with what we saw for the overall distributions (Table 5-5).

55% of the opens were read-only with most of the read-only opens going to perm files (Table 5-33). Users showed a strong tendency to open perm files read-only and other files write-only or

cut	opens	% opens	files	% files	opens/file
U+file_LOG	837	0.8%	101	0.3%	8.3
U+file_PERM	65051	62.4%	8662	29.0%	7.5
U+file_TEMP	38420	36.8%	21127	70.7%	1.8
U	104308	100%	29890	100%	3.5

Table 5-32: User opens to user files

cut	read-only		write-only		read/write		total
	opens	fraction	opens	fraction	opens	fraction	opens
U+file_LOG	117	14.0%	623	74.4%	97	11.6%	837
U+file_PERM	50193	77.5%	13296	20.5%	1310	2.0%	64799
U+file_TEMP	7349	19.1%	19891	51.8%	11140	29.0%	38380
U	57659	55.4%	33810	32.5%	12547	12.1%	104016

Table 5-33: Modes of open for user open-close sessions to user files

category	opens	% opens	files	% files	opens/file
library	2036	3.1%	91	1.1%	22.4
manual pages	776	1.2%	181	2.1%	4.3
program source	10538	16.2%	1486	17.1%	7.1
includes	3093	4.8%	306	3.5%	10.1
objects	5617	8.6%	467	5.4%	12.0
personal configuration	20278	31.2%	1638	18.9%	12.4
mail spool	2049	3.1%	453	5.2%	4.5
other	20644	31.8%	4040	46.6%	5.1

Table 5-34: Function of opened user perm files

read/write.

30% of the activity to perm files was to program development files ("program source," "includes," and "objects" in Table 5-34). A similar number of references were to personal configuration files (often referred to as "dot files"). Most of the rest of the references were unidentifiable.

5.5.3.2. Per Open Results for User Files

User open activity to user files (Figure 5-25) showed a busy period during the work day, with activity tapering off in the late evening. This is typical of a university environment. There was some early morning activity due to user background jobs. The overall level of activity was much

cut	reads		writes		overall (r+w)	
	bytes/sec	fraction	bytes/sec	fraction	bytes/sec	fraction
U+file_LOG	9.6	1.0%	4.6	1.1%	14.2	1.0%
U+file_PERM	401	41.0%	121	28.6%	522	37.3%
U+file_TEMP	568	58.0%	297	70.4%	865	61.7%
U	978	100%	423	100%	1401	100%
no cut (Table 5-8)	4190	-	800	-	4990	-

Table 5-35: Bytes read/written by users to user files

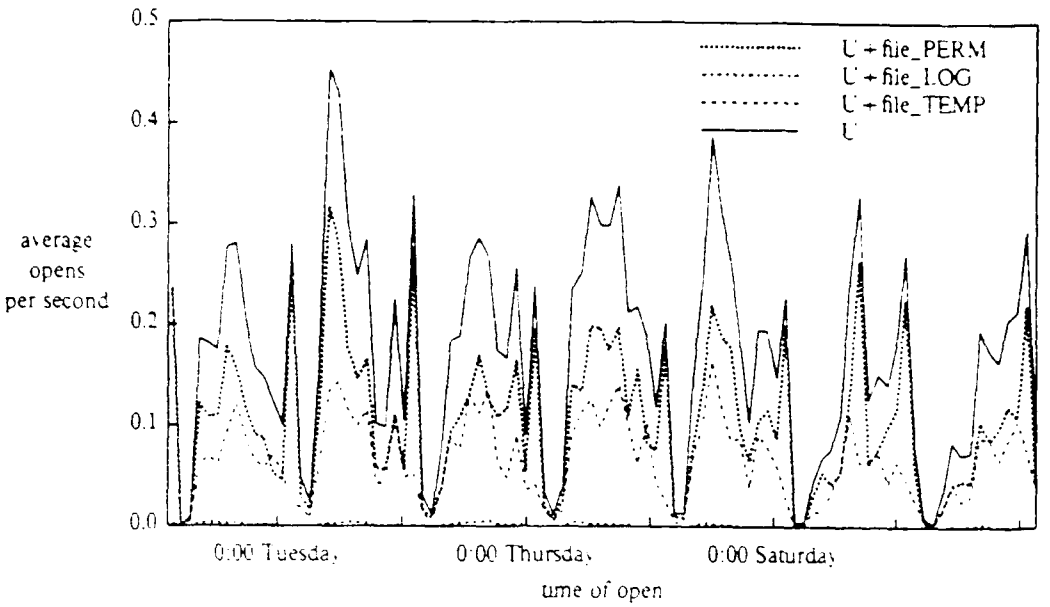


Figure 5-25: Average number of file opens per second (2 hour resolution, U cut)

less than what we saw for the system as a whole (user opens to user files accounted for 14% of the open activity) and was generally less bursty.

User reads and writes to user files accounted for 28% of the bytes transferred during the logging period. Most of the transfers (61.7%) were to and from temp files (Table 5-35). Few bytes were transferred to or from log files.

distribution	min	max	mean	median	std deviation
U+file_LOG, dynamic	0	1.28e6	90400	39000	1.7e5
U+file_PERM, dynamic	0	2.49e6	6205	1230	4.0e4
U+file_TEMP, dynamic	0	1.30e6	5006	310	2.4e4
U, dynamic	0	2.49e6	6440	930	3.9e4
all, dynamic (Table 5-9)	0	2.49e6	18800	710	6.2e4
all, static	0	7.95e6	8020	1600	5.6e4

Table 5-36: User file size distributions

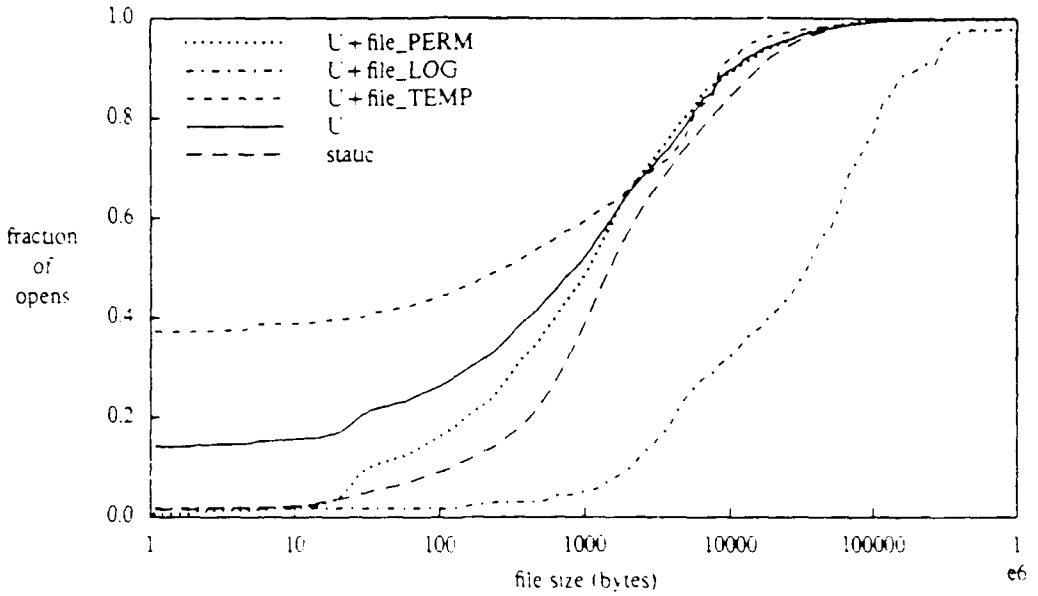


Figure 5-26: Dynamic file size distributions (cumulative, measured at close, U cut)

Cumulative file size distributions for users files, weighted by the number of user opens and cut by the file class, are given in Figure 5-26 and Table 5-36. Referenced user files were, on average, smaller than other referenced files. This was due, in part, to the large number of zero length temp files and to the absence of the large, frequently accessed administration files seen in the overall data.

Users accessed most of their files completely (Figures 5-27 through 5-30 and Tables 5-37 through 5-40). 90% of opens with read access (read-only or read/write) resulted in the file being completely read (compared to 68% for the system as a whole). 83% of files opened with write access were completely written (compared to 78% for the system as a whole). Nearly all files opened read-only were completely read.

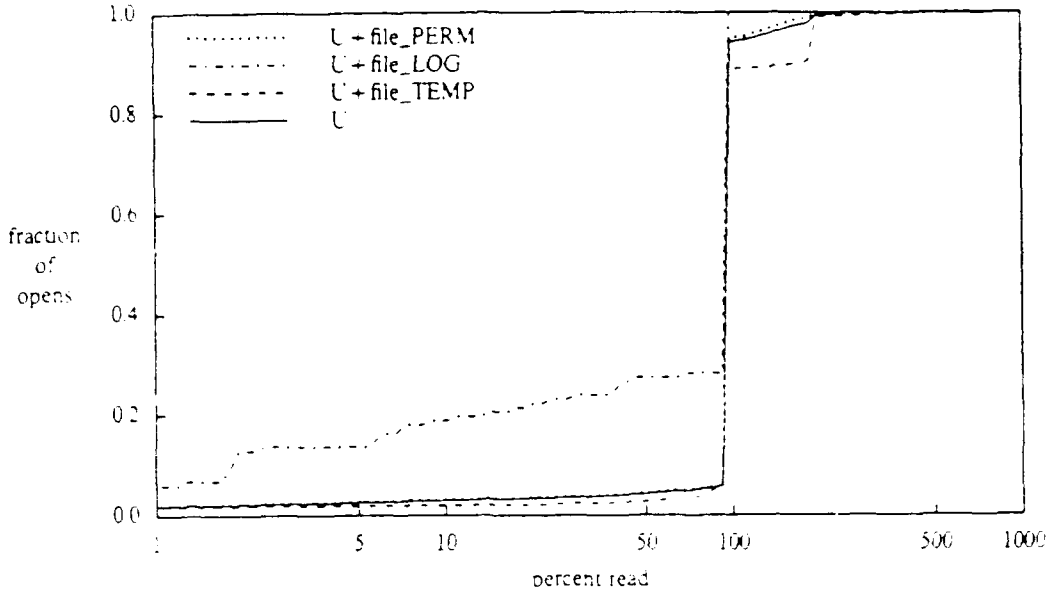


Figure 5-27: Percent of file read for read-only opens (cumulative, U cut)

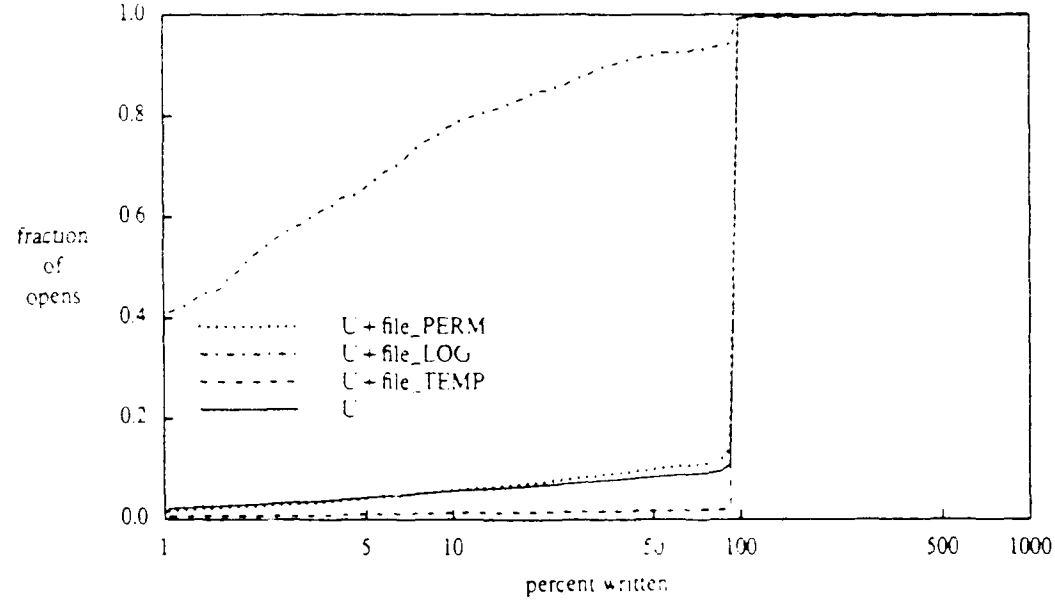


Figure 5-28: Percent of file written for write-only opens (cumulative, U cut)

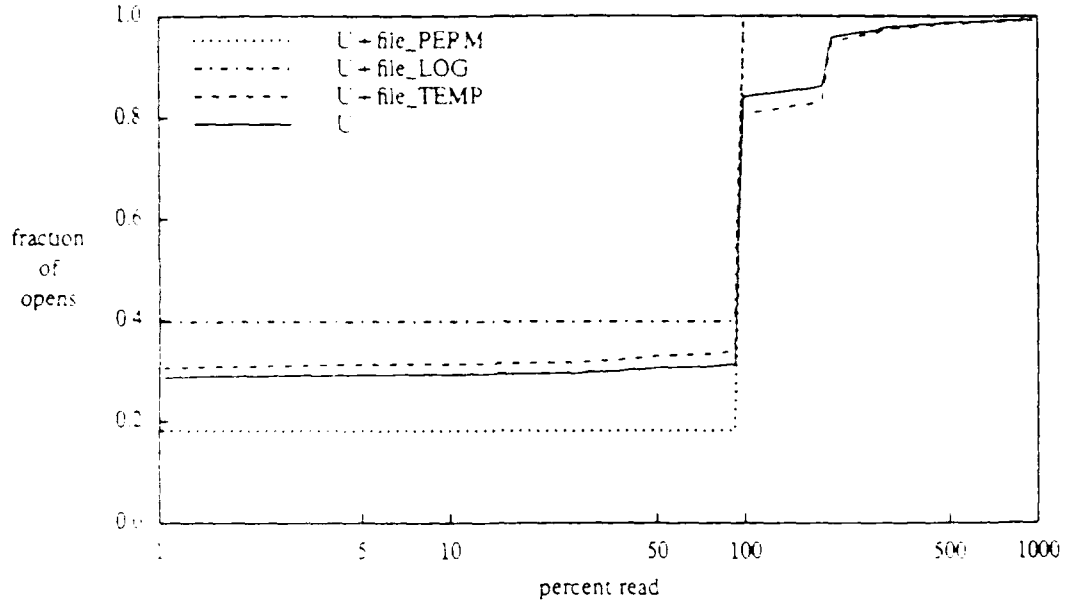


Figure 5-29: Percent of file read for read/write opens (cumulative, U cut)

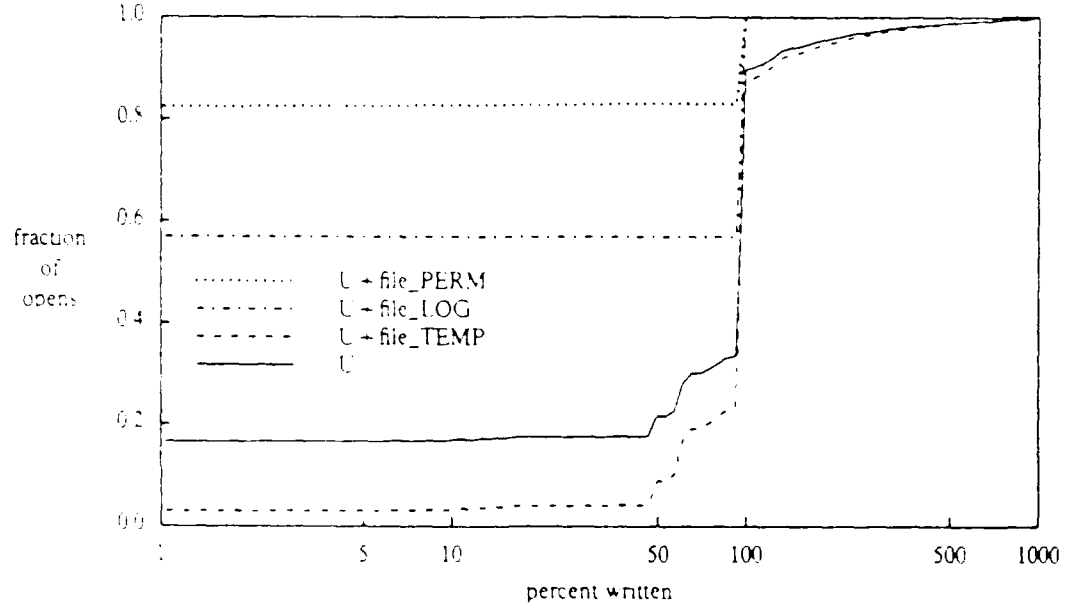


Figure 5-30: Percent of file written for read/write opens (cumulative, U cut)

distribution	min	max	mean	median	std dev	<100%	>100%
U+file_LOG	0	100	75.3	100	40	28%	0%
U+file_PERM	0	12500	99.9	100	110	5.7%	5.2%
U+file_TEMP	0	1160	109	100	47	7.2%	11.4%
U	0	12500	100.9	100	104	5.9%	6.0%
no cut (Table 5-11)	0	64100	83.9	100	202	31%	2.9%

Table 5-37: Percentage of users files read (read-only opens)

distribution	min	max	mean	median	std dev	<100%	>100%
U+file_LOG	0	100	12.4	1.9	26	94%	0%
U+file_PERM	0	200	90.8	100	27	14%	0.8%
U+file_TEMP	0	9600	106.4	100	201	2.0%	0.7%
U	0	9600	95.6	100	134	11%	0.8%
no cut (Table 5-12)	0	9600	85.7	100	53	15%	0.1%

Table 5-38: Percentage of user files written (write-only opens)

distribution	min	max	mean	median	std dev	<100%	>100%
U+file_LOG	0	100	60.2	100	49	40%	0%
U+file_PERM	0	100	81.8	100	39	18%	0%
U+file_TEMP	0	65000	159	100	1670	34%	19%
U	0	65000	145	100	1500	31%	16%
no cut (Table 5-13)	0	65000	82.9	100	615	40%	11%

Table 5-39: Percentage of user files read (read/write opens)

distribution	min	max	mean	median	std dev	<100%	>100%
U+file_LOG	0	100	43.0	<1	50	57%	0%
U+file_PERM	0	100	17.0	<1	37	83%	1%
U+file_TEMP	0	3600	112	100	156	23%	12%
U	0	3600	95.7	100	147	34%	10.3%
no cut (Table 5-14)	0	20000	51.8	<1	249	61%	2.7%

Table 5-40: Percentage of user files written (read/write opens)

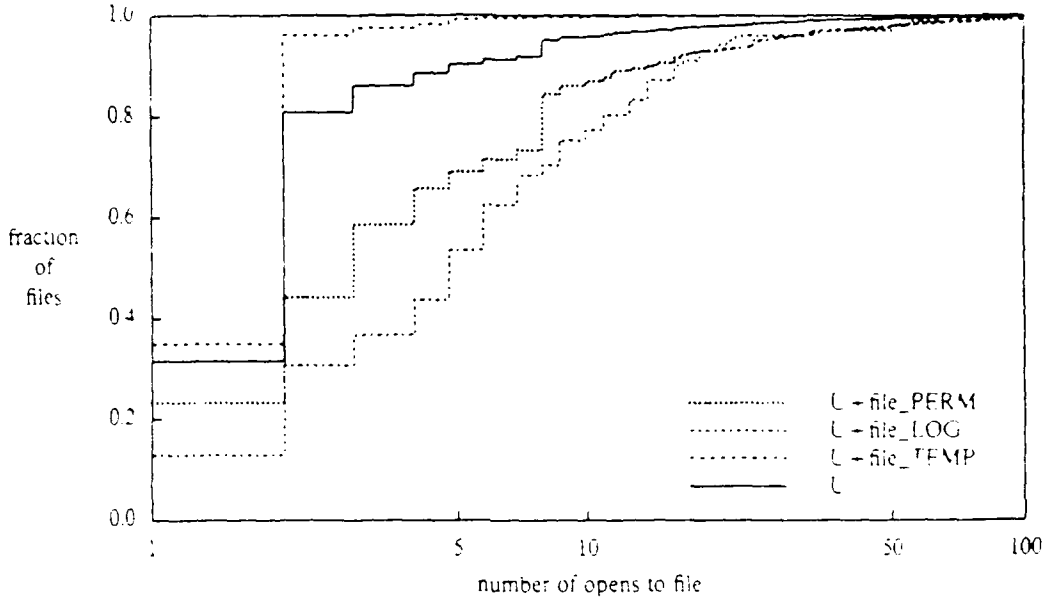


Figure 5-31: Number of opens per active file (cumulative, U cut)

distribution	mean	median	opened once	opened twice	opened more than twice	max
U+file_LOG	8.3	5	13%	18%	69%	79
U+file_PERM	7.5	3	23%	21%	56%	562
U+file_TEMP	1.8	2	35%	61%	3.9%	198
U	3.5	2	31%	50%	19%	562
no cut (Table 5-17)	7.5	2	48%	33%	19%	26800

Table 5-41: Number of user opens/user file

5.5.3.3. Per File Results for User Files

User temp files were generally accessed twice. User log and perm files saw somewhat more activity (Figure 5-31 and Table 5-41). Although only 19% of the user files seen were referenced more than twice during the week of logging, these files accounted for 63% of the opens. User file distributions don't show the frantic activity to a few files that we saw for the overall distribution, but there was still a small group of relatively active files that accounted for the majority of the

opens.

Interopen intervals for user files (Figure 5-32 and Table 5-42) bore little resemblance to the results we saw for the overall data. Intervals for user files could generally be expressed in minutes instead of seconds. Temp files were an exception here. The second open to a temp file usually followed immediately after the first one.

File and data lifetimes for user files are shown in Figures 5-33 and 5-34. Most user perm and log files had lives exceeding our logging period. Data in user log and perm files were also long lived

distribution	min	max	mean	median	std deviation
U+file_LOG	0.02	5.4e5	31100	3100	5.7e5
U+file_PERM	0	5.4e5	21400	450	4.1e4
U+file_TEMP	0.01	4.2e5	1390	0.38	1.2e4
U	0	5.4e5	16900	120	3.7e4
no cut (Table 5-21)	0	5.4e5	7502	60	2.2e4

Table 5-42: User file interopen intervals (seconds)

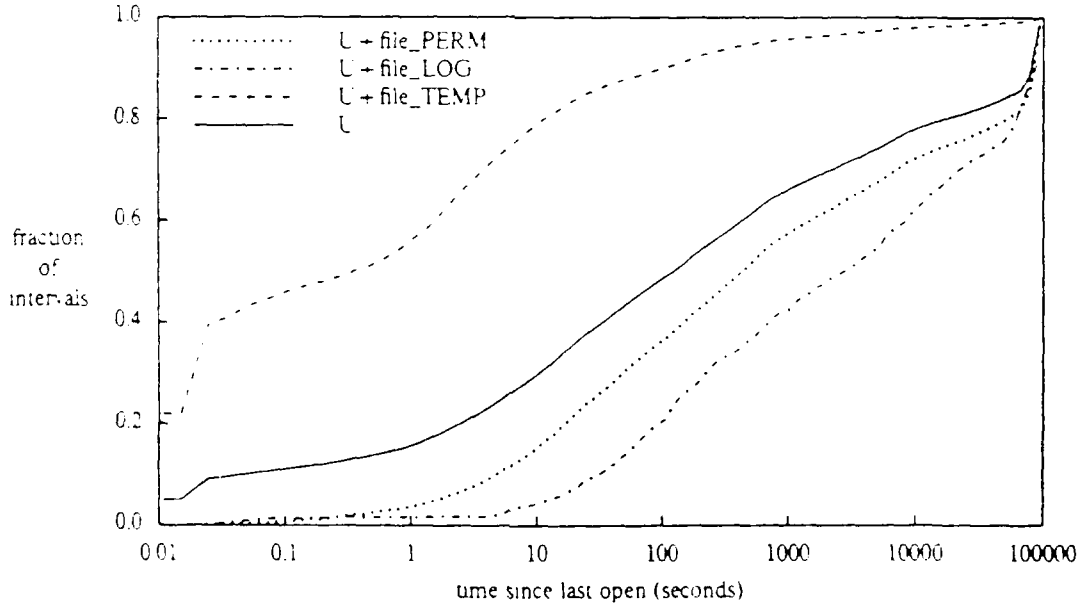


Figure 5-32: File interopen intervals (cumulative, U cut)

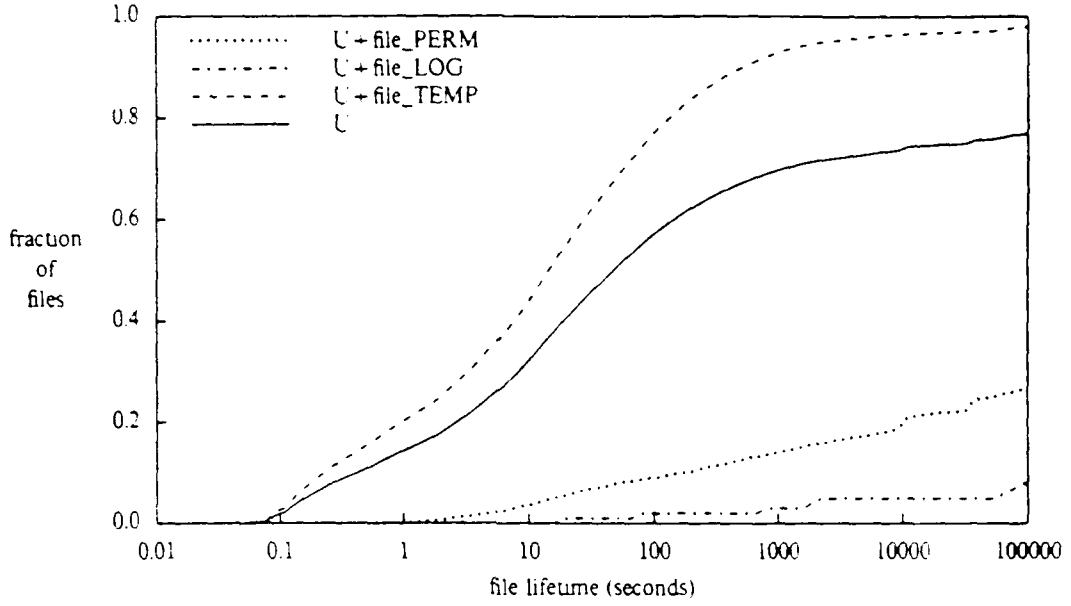


Figure 5-33: File lifetimes (cumulative, U cut)

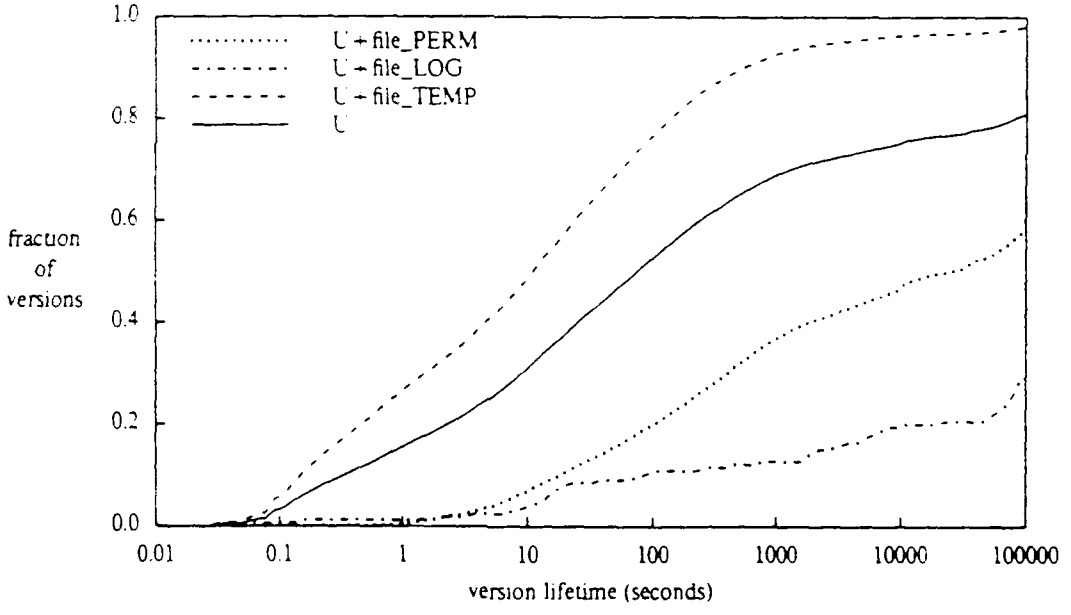


Figure 5-34: Version lifetimes (cumulative, U cut)

(this was not the case for the overall data). Half of all user temp files lived less than 15 seconds.

Tables 5-43 and 5-44 provide some statistics on user sharing of user files. Sharing was restricted to log and perm files. The low mean number of inversions (0.069) indicates that sharing was

incidental to the normal use of user files.

5.6. Summary

This chapter has described in detail the collection and analysis of short term file reference data from a 4.2BSD UNIX system supporting university research. As is true with all studies of this sort, our results can be guaranteed to be valid only for our system at the time of data collection. Care should be taken in applying the results to other situations. Given this caveat, our major findings

cut	readers		writers		users (r | w)		inversions	
	mean	>1	mean	>1	mean	>1	mean	max
U+file_LOG	0.634	3.0%	0.99	3.0%	1.099	5.9%	0.356	9
U+file_PERM	0.936	2.7%	0.632	2.1%	1.12	4.9%	0.231	163
U+file_TEMP	0.798	0.02%	0.995	0%	1.0	0.04%	0	2
U	0.838	0.80%	0.889	0.63%	1.035	1.5%	0.069	163
no cut (Table 5-22)	0.792	6.6%	0.930	2.4%	1.30	11.8%	1.16	12529

Table 5-43: User file sharing

number	readers		writers		users (r | w)		inversions	
	count	cum	count	cum	count	cum	count	cum
0	5367	18.0%	3826	12.8%	-	-	29452	98.5%
1	24284	99.20%	25877	99.37%	29452	98.5%	198	99.20%
2	146	99.69%	98	99.70%	255	99.39%	109	99.56%
3	45	99.84%	28	99.80%	74	99.64%	27	99.65%
4	21	99.91%	19	99.86%	40	99.77%	23	99.73%
5	11	99.95%	8	99.89%	19	99.83%	14	99.78%
6	2	99.95%	7	99.91%	9	99.86%	7	99.80%
7	3	99.96%	7	99.93%	10	99.90%	5	99.82%
8	2	99.97%	3	99.94%	5	99.91%	7	99.84%
9	2	99.98%	5	99.96%	7	99.94%	7	99.86%
10	0	99.98	2	99.97%	2	99.94%	4	99.88%
>10	7	100%	10	100%	17	100%	37	100%
total	29890	-	29890	-	29890	-	29890	-

Table 5-44: readers, writers, users and inversions; user references to user files

are as follows:

(1) Opened files in our environment are small, with half being under 710 bytes long.

(2) The majority of bytes read come from larger files (greater than 20,000 bytes long).

(3) 68% of files opened with read access are completely read and 78% of files opened with write access are completely written. The percentage read and written depends strongly on the class of the file (log, perm or temp), the mode of open, the file opener and the size of the file. In particular, log files are almost never completely written and users completely read 94% of files they open read-only.

(4) Temporary files are usually accessed only once or twice and most live for less than a minute. Log and permanent files live for much longer periods and see more open activity.

(5) Most opens go to files opened hundreds or thousands of times a week. Large administrative files account for a substantial fraction of this activity.

(6) Files are generally open for only a few tenths of a second.

(7) Interopen intervals in our environment are short. Half are under 60 seconds. The interopen interval depends strongly on the class (log, permanent or temporary) and owner of the file.

(8) Most sharing is restricted to system and net files in our environment. Sharing of user files is incidental to their normal use.

(9) Executed files are relatively large (half are over 20,000 bytes), heavily used and few in number.

(10) Half of all execute requests go to a very small number of executable files (13 files; 2.2% of the referenced executables).

(11) We see substantial differences in file access patterns based on the class of the file, the owner of the file and the class of the file opener. In particular, overall reference patterns do not match user file reference patterns and reference patterns for logs, permanent files and temporary files bear little resemblance to each other.

These results have a number of interesting implications for DFS design. These implications will be explored in Chapter 7.

Chapter 6

Directory Reference Patterns in a UNIX Environment

6.1. Introduction

This chapter continues the analysis of UNIX file system reference patterns that we began in Chapter 5. In that chapter we focussed on file reference patterns. This chapter examines directory reference patterns in some detail.

Our study of directory reference patterns is motivated by studies that found that 40% of BSD UNIX system call overhead was due to name resolution [Leffler 84], that half of all network traffic in LOCUS was in support of name resolution [Sheltzer 86], and by our file reference results showing that most referenced files were small enough to read in a single disk access.

Chapter 5 described modifications that were made to the UNIX kernel to collect a log of accesses to files. This log includes a complete record of the paths used to create, open, execute and delete files, to open directories, and to create, delete, and modify directories. This chapter uses that path information to examine the overhead of name resolution in accessing files, directory read/write ratios, the rate of change of directory nodes, and directory sharing. As in Chapter 5, we have tried

to present the information in a way that gives a qualitative feel for the way that directories are used on our UNIX system.

Section 6.2 describes aspects of our data collection methodology that are important in analyzing directory reference patterns. Section 6.3 outlines the approach we used in analyzing the data. Section 6.4 presents the results of this analysis. Section 6.5 summarizes our results.

6.2. Data Collection Methodology

Section 5.2 described the method we used to generate a trace of file references. This trace includes, among other things, a complete record of the paths used to create, open, execute and delete files, to open directories, and to create, delete, and modify directories. Our original purpose in collecting data was to track references to files. Because of this, some calls that cause directories to be referenced (for name resolution) were omitted from the log. These calls were:

(1) Protection: chmod, chown.

(2) Status: readlink, lstat, stat, utimes, access.

(3) Administrative: acct, mknod, mount, setquota.

(4) UNIX domain IPC (side effect of the 4.2BSD implementation): bind, connect.

Of these calls, only *lstat* and *stat* are likely to occur with any frequency. These two calls retrieve status information on a file (size, protection, access date, and so on) from the file inode. Mogul found [Mogul 86b], in studying another 4.2BSD system, that lstat and stat were used nearly twice as often as the calls that we logged. Most of these status calls were made, though, by an administrative process that scanned the entire file system on a regular basis. A similar program is run on our system, but only half as often and on a smaller, busier file system. Based on this, we estimate that lstat and stat calls occur about half as often as open and creat calls on Seneca. Further, these status calls are generally tightly clustered in time (most will occur during the 4AM scan of the file system) and so we expect that they will have little effect on the results we will be presenting.

Another potential contribution to directory references that we have not logged is from system calls that fail. We logged only successful calls.

6.3. Analysis Method

6.3.1. Conventions

Our method for analyzing directory reference patterns is similar to the method used for file reference (section 5.4). There are a few conventions, observed by all analysis programs, that are specific to the analysis of directory references:

(1) All of the analysis presented here is at the *node* (entire directory) level. We haven't attempted to analyze references to individual directory entries or pages. See Chapter 5 (particularly the file interopen interval and lifetime distributions) and [Leffler 84] for information on individual entries.

(2) Directory sizes include entries for "." (the directory itself) and ".." (the parent). These entries are always present in a UNIX directory and so the minimum directory size is 2 entries.

(3) Directory sizes given in bytes or blocks assume that the 4.2BSD directory layout is used (that is, an 8 byte header, space for the name itself, and a 1 byte trailer, padded out to a 4 byte boundary) and that there are no "empty" entries. This last assumption means that we probably understate somewhat the number of blocks required to read a directory.

(4) All component resolutions are marked as having taken place at the time the system call being analyzed finished. In real life, of course, these resolutions won't occur simultaneously. Because of this, intervals of less than 100ms should not be taken seriously.

(5) When we encounter a record in the log that contains a path to resolve, we take each path component in turn, resolving it individually. Each directory used in the resolution is marked as having been referenced and the appropriate histograms are

incremented. Resolution starts either at the root of the file system (for an absolute path) or in the current working directory of the process that generated the record. So, for example, an OPEN record that specifies a path of "/u/rick/.login" generates 3 directory references: to "/" to resolve "u"; to "u" to resolve "rick"; and to "rick" to resolve ".login."

(6) All components in a path are resolved. No attempt is made to short-circuit degenerate components or path segments. So, for example, resolving "./.login" requires two references (one for "." and one for ".login") and resolving "/u/rick/.login" would require 3 references even if the working directory of the process making the request is "/u/rick." This is consistent with the approach used by 4.2BSD.

6.3.2. Cuts

One expects that directory reference patterns will be different for user versus system processes, user versus system directories, batch versus interactive work, and so on. This was certainly the case for file references (Chapter 5). To investigate these effects, we use the three basic types of cuts described in section 5.4.2 (on process owner, object owner, and file type). In addition, we use a cut based on the *UNIX file system* of the referenced directory. The overall file name space on a UNIX machine is actually made up of a number of physical file systems that form subsets of the overall naming tree. There were 5 physical file systems on Seneca at the time data was collected, mounted as follows: "/" (the root of the file system), "/u" (user files), "/usr" (system files), "/usr/spool" (USENET news and spooling space for printers and UUCP), and "/tmp" (scratch space).

These cuts may be combined to give other more specific cuts. 9 cuts are used in this chapter (the first 8 cuts were also used in Chapter 5 and are described in detail in section 5.4.2):

(1) **no cut**: This cut passes all records in the log to the user analysis routines.

(2) **ruid_NET**: Passes references by *net* processes.

(3) **ruid_SYSTEM**: Passes references by system processes.

(4) **ruid_USER**: Passes references by processes running under user accounts.

(5) **dir_owner_NET**: Passes references to directories owned by UUCP, USENET news, and notes accounts.

(6) **dir_owner_SYSTEM**: Passes references to directories owned by system accounts.

(7) **dir_owner_USER**: Passes references to user directories.

(8) **owner_USER+ruid_USER**: Passes references made by user processes, but only if the leaf object is owned by a user. This gives a trace of directories accessed in resolving user references to user files.

(9) **ruid_USER+/u**: Passes references made by user processes to directories in the /u file system (this is the file system on Seneca that holds all user directories). While the owner_USER+ruid_USER cut includes all directories that are referenced in accessing user files (including, for example, "/tmp" for temp files and "/" for absolute path names), the ruid_USER+/u cut only includes that subset of files and directories on /u. This cut will be of particular interest to DFS designers who combine a global user file space with local system directories [Satyanarayanan 85].

6.4. Directory Reference Patterns

This section presents the results of our analysis. A full analysis of the data was done using 21 different cuts (the 9 cuts listed in section 6.3.2 plus cuts on other file systems and on ruid, file system, and directory owner combinations). It is clearly impractical to present results for the full set of cuts. We have generally included only those tables and histograms that are particularly characteristic or striking.

Overall system activity and per call results are presented using ruid cuts. These cuts show the overall contribution from each of the user classes and point out some of the differences in the way that these classes use the file system. Analysis that concentrates on individual directories is presented using directory owner cuts. Directory owner cuts show us roughly where in the file system activity is concentrated and allow us to investigate the activity on a directory-by-directory

basis.

In some cases we give more detailed results on user activity using the owner_USER+ruid_USER and ruid_USER+/u cuts. These cuts give us a data sample that allows us to investigate reference patterns that a DFS dealing primarily or wholly with user files would see.

6.4.1. Basic Statistics

Table 6-1 gives a summary of records collected for events that referenced directories (this is a subset of the records shown in Table 5-3). The first 3 columns give the number of records of each type collected, the average rate for that type of record, and the percentage of collected records that this represents. The remaining columns show the number of records collected cut by the ruid of the calling process and the percentage of the total for the ruid class. Opens accounted for 2/3 of the path requests we logged. Chdir, unlink, and execute calls accounted for most of the rest of the requests. There were relatively few directory structure modification requests.

record	no cut			ruid_NET		ruid_SYSTEM		ruid_USER	
	count	per hr	fraction	count	fraction	count	fraction	count	fraction
mkdir	936	5.5	0.07%	795	0.19%	2	0%	139	0.03%
rename	3211	19	0.23%	1946	0.46%	408	0.08%	857	0.19%
rmdir	913	5.4	0.06%	780	0.19%	0	-	133	0.03%
symlink	16	0.1	0%	0	-	3	0%	13	0%
chdir	136063	806	9.7%	19102	4.5%	71854	13.5%	45106	10.0%
chroot	0	-	-	0	-	0	-	0	-
execute	125064	741	9.0%	26761	6.4%	38093	7.2%	60209	13.3%
link	42929	254	3.1%	25694	6.1%	7301	1.4%	9934	2.2%
open	965087	5720	68.7%	277350	65.9%	393661	74.1%	294070	64.9%
truncate	0	-	-	0	-	0	-	0	-
unlink	130929	776	9.3%	68342	16.2%	19861	3.7%	42726	9.4%
total	1405148	8323	100%	420770	100%	531183	100%	453187	100%

Table 6-1: Records logged

Opens may be further broken down by the type of object being opened (Table 6-2). While most requests were to open regular files, there were also a significant number of directory opens. Processes open directories in UNIX to scan the contents (as opposed to resolving a single name). This is commonly done by user processes to satisfy interactive requests to list directory contents. Directory open activity by system processes was due to daemons examining spool directories for work and to housekeeping scans of the file system. In our analysis we have counted the open of a directory as a single reference to the directory.

Each of our ruid classes accounted for roughly 1/3 of the path resolution requests (Table 6-3). Most of these paths were specified absolutely (that is, were resolved starting at the root of the naming tree). Overall, only about a quarter of the objects being referenced were listed in the working directory of the process making the request. This is a reflection, in part, of the high level of activity to system files. As we saw in Chapter 5, over half of all file opens went to system files. 4.2BSD makes heavy use of system files to store system configuration and status information. Since these files are often opened as an incidental part of other activity, they are usually not in the current working directory and so are referenced absolutely.

type	no cut		ruid_NET		ruid_SYSTEM		ruid_USER	
	opens	fraction	opens	fraction	opens	fraction	opens	fraction
regular file	754285	78.2%	249825	90.1%	298186	75.7%	206268	70.1%
directory	170448	17.7%	17275	6.2%	72625	18.4%	80548	27.4%
block special	922	0.1%	0	-	60	0.02%	862	0.3%
character special	39432	4.1%	10250	3.7%	22790	5.8%	6392	2.2%
total	965087	100%	277350	100%	393661	100%	294070	100%

Table 6-2: Opens, by object type

cut	paths	% paths	absolute path	leaf in working dir
ruid_NET	4.48e5	30.8%	74.2%	17.1%
ruid_SYSTEM	5.39e5	37.0%	70.9%	28.3%
ruid_USER	4.69e5	32.2%	66.6%	32.8%
owner_USER+ruid_USER	2.55e5	17.5%	54.3%	45.9%
ruid_USER+/u	1.57e5	10.8%	35.2%	64.7%
no cut	1.46e6	100%	70.5%	25.9%

Table 6-3: Path statistics

Our compound user cuts eliminate this activity to system files. If we look at just user activity to files on /u (the user file system), we find that 2/3 of the paths we saw specified an object in the working directory of the process making the request. Note, though, that user references to objects on /u accounted for only a third of the overall user paths and a tenth of the system activity. For references by users to all user objects (those on /u plus user files in shared system directories such as /tmp and /usr/spool/mail), the fraction of paths specifying an object in the working directory drops to less than 1/2.

Each path resolved may (and usually does) have more than one component. Table 6-4 gives some information on the number of components per path for each of our ruid classes. Note that each path had, on average, almost 3 components, each of which required a directory reference to resolve. Paths for net processes were particularly long. This was caused by the relative depth of the net directory trees (rooted in /usr/spool/news, /usr/spool/uucp, and so on) coupled with the heavy use of absolute path names by net processes.

User paths specifying user objects were generally shorter, with an average of slightly more than two name components to resolve per path. If the target object was on /u, an average of 1.57 of the components resolved were on the /u file system (note that this doesn't include references to "/" for absolute paths, since "/" is not on the /u file system).

cut	mean	median	1 component	2	3	4	>4	max
ruid_NET	3.45	4	8.0%	34.9%	6.3%	25.7%	25.2%	8
ruid_SYSTEM	2.48	2	28.4%	31.0%	6.3%	33.5%	0.7%	8
ruid_USER	2.22	2	32.5%	34.7%	19.2%	8.9%	4.7%	11
owner_USER+ruid_USER	2.11	2	42.2%	25.3%	18.2%	10.5%	3.8%	11
ruid_USER+/u	1.57	1	61.9%	26.7%	6.5%	2.7%	2.1%	9
no cut	2.70	2	23.4%	33.4%	10.5%	23.2%	9.5%	11

Table 6-4: Components/path

cut	total		reads		writes		reads/ writes
	references	fraction	references	fraction	references	fraction	
ruid_NET	1.59e6	37.5%	1.44e6	36.4%	1.42e5	52.1%	10.1
ruid_SYSTEM	1.48e6	34.9%	1.44e6	36.4%	4.03e4	14.8%	35.7
ruid_USER	1.17e6	27.6%	1.08e6	27.2%	9.01e4	33.1%	12.0
dir_owner_NET	7.42e5	17.5%	6.17e5	15.6%	1.25e5	46.1%	4.9
dir_owner_SYSTEM	3.09e6	73.0%	2.96e6	74.7%	1.23e5	45.4%	24.1
dir_owner_USER	4.06e5	9.6%	3.83e5	9.7%	2.32e4	8.6%	16.5
owner_USER+ruid_USER	6.16e5	14.5%	5.36e5	13.5%	7.99e4	29.4%	6.7
ruid_USER+/u	3.07e5	7.2%	2.86e5	7.2%	2.20e4	8.1%	13
no cut	4.24e6	100%	3.96e6	100%	2.72e5	100%	14.6

Table 6-5: Reference statistics

While each of the ruid categories accounted for a roughly equal number of references, nearly 3/4 of all references went to system directories (Table 6-5). This is not surprising, since most references were absolute and so this implies that even references to files in user or news subtrees often required two or three system directories to resolve. There was relatively little activity to user directories. Overall, 93.4% of the references were directory reads and 6.6% were directory

writes[1]. Nearly half of the writes were to system directories (mostly to /tmp and /usr/spool/mqueue), with most of the rest going to net directories. Net directories were not heavily used overall, but had a particularly low read/write ratio and so a high fraction of the writes.

User references to their files and directories accounted for only 14.5% of the references on the system and about half of the references made by users. Relatively few user writes were to user directories. As we will see in section 6.4.3, most were to system temporary and spool directories.

Most directories belonged to users (Table 6-6) but, as we saw above, there was relatively little activity to these directories. Again, this is a reflection of the heavy activity to system directories.

6.4.2. Per Reference Results

The directory reference activity over time is shown in Figure 6-1. References followed a daily pattern with a busy period between 9am and 6pm, overlaid by bursts from net activity (news reception) and a strong peak in the early morning (news expiration and the housekeeping scan of the file system). Weekends were relatively quiet. Except for the strength of the early morning

cut	directories	% directories	references/directory
dir_owner_NET	1275	23.5%	582
dir_owner_SYSTEM	427	7.9%	7230
dir_owner_USER	3713	68.6%	109
no cut	5415	100%	782

Table 6-6: References/directory

[1]Note that directory writes in UNIX represent changes in the naming tree (such as adding and deleting files). Information on the objects named (size, last use and so on) is kept elsewhere.

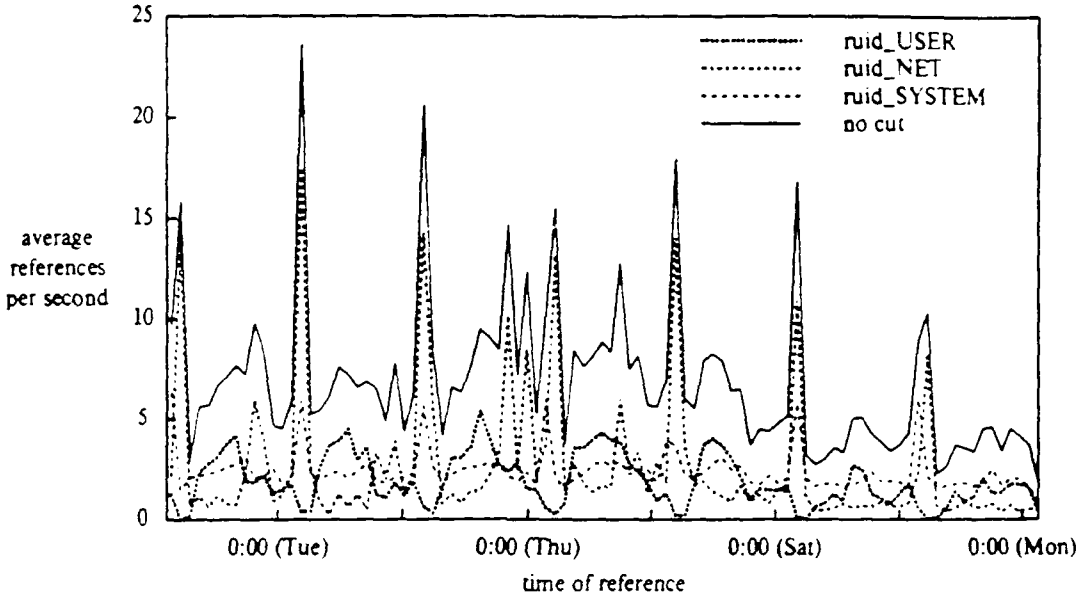


Figure 6-1: Directory references per second (~2 hour resolution)

peak, this pattern follows closely the one we saw in Chapter 5 for file opens. The relative strength of the morning peak is due to the long length of paths used by net processes and the inclusion of directory opens (the primary housekeeping activity we logged).

User activity to user directories (Figure 6-2) showed a busy period during the day, with activity tapering off in the late evening. This is typical of a university environment. There was some early morning activity due to user background jobs.

Figure 6-3 shows the size (in entries) of referenced directories, weighted by the number of references made and cut by the owner of the referenced directory. Note that these are *cumulative* distributions. At any point on a curve, the y value is the fraction of directories with sizes less than or equal to the x value. For comparison purposes, we have included here the static directory size distribution (this is the distribution that would result if each directory on the system were referenced once). Table 6-7 gives some statistics on these distributions.

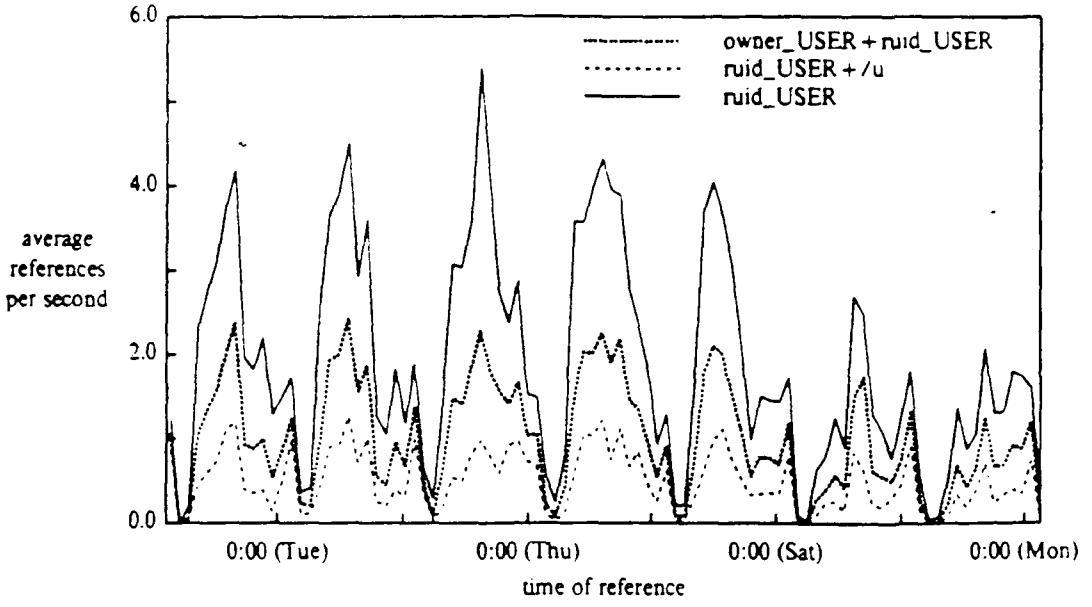


Figure 6-2: Directory references per second (~2 hour resolution, user cuts)

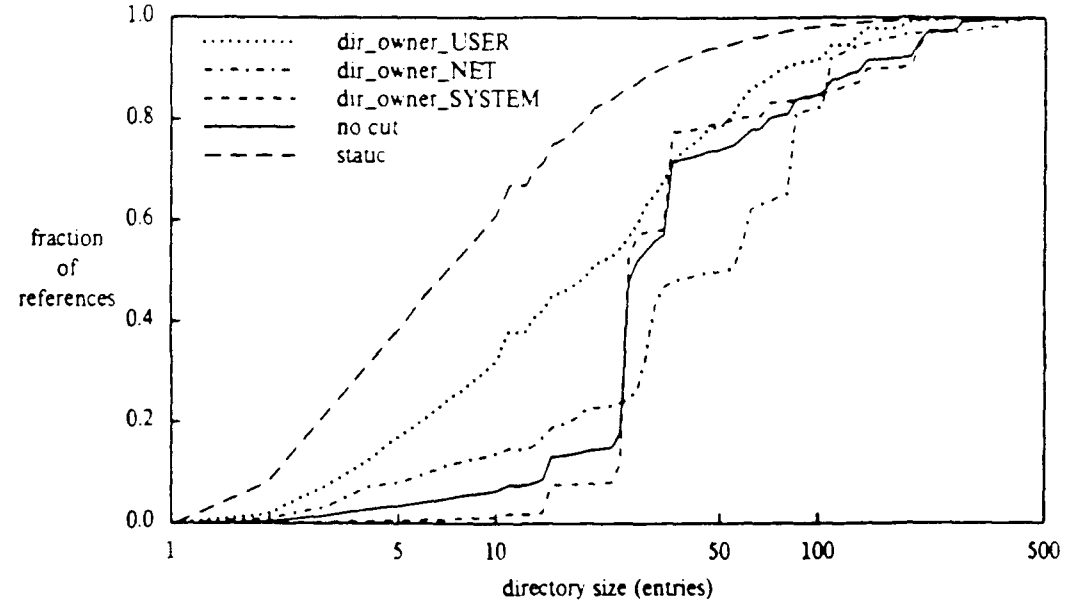


Figure 6-3: Size of referenced directories (cumulative, in entries)

distribution	min	max	mean	median	std deviation
dir_owner_NET, dynamic	2	500	63.1	51	65.3
dir_owner_SYSTEM, dynamic	2	471	55.1	26	63.3
dir_owner_USER, dynamic	2	327	34.2	20	40.2
owner_USER+ruid_USER	2	484	60.8	29	64.2
owner_USER+/u	2	282	82.9	40	80.6
ruid_USER	2	484	66.0	33	69.8
all, dynamic	2	500	54.5	26	62.2
all, static	2	471	15.8	8	27.3

Table 6-7: Directory size distributions (in entries)

From Figure 6-3 we see that most directories on Seneca were small (half had under 8 entries). Referenced directories were considerably larger (median of 26 entries), but still small by most standards. Since "/," "/usr," and "/usr/spool" had 26, 35 and 26 entries respectively and accounted for nearly half of the references, this result was inevitable. The median size of 26 entries implies that, in the absence of other factors, the median number of comparisons needed to resolve a component was 13. This agrees with 4.2BSD measurements done elsewhere [Mogul 86a]. The small static median is also typical of 4.2BSD systems [Mogul 86b]. These distributions all have long tails and so the means are considerably higher.

Directories on /u referenced by users (weighted by the number of references) were generally somewhat larger than referenced directories on the system as a whole (Figure 6-4 and Table 6-7). This was due, in part, to the relatively high level of activity to /u (at 200 entries) and the absence of the heavily referenced system directories (at 26 entries).

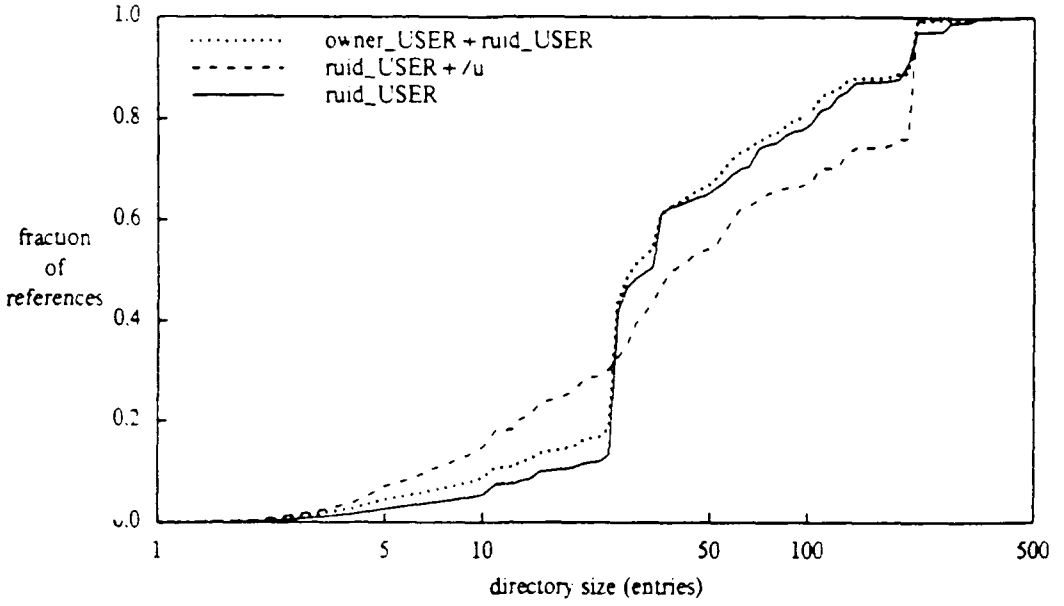


Figure 6-4: Size of referenced directories (cumulative, in entries, user cuts)

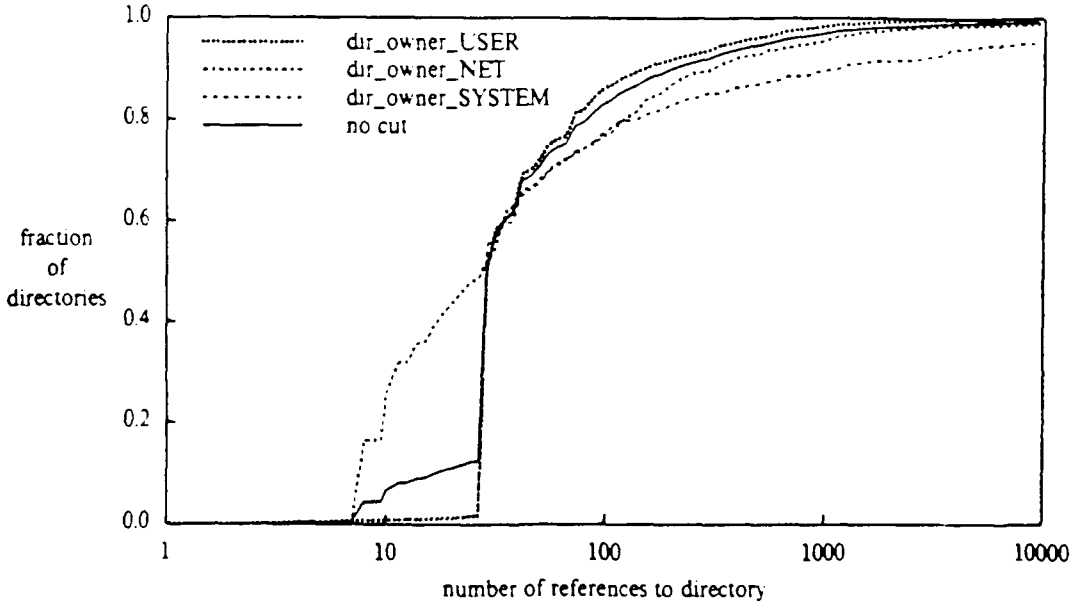


Figure 6-5: Number of references per directory (cumulative)

distribution	min	max	mean	median	std deviation
dir_owner_NET	8	1.45e5	582	28	5.9e3
dir_owner_SYSTEM	28	1.06e6	7230	28	6.3e4
dir_owner_USER	2	1.38e4	109	28	4.1e2
owner_USER+ruid_USER	1	1.44e5	327	12	4.0e3
ruid_USER+/u	1	7.18e4	219	18	2.0e3
ruid_USER	1	3.24e5	582	20	8.1e3
no cut	2	1.06e6	782	28	1.8e4

Table 6-8: Number of references/directory

6.4.3. Per Directory Results

The number of references to a directory over a period gives an indication of the potential benefits of caching or, for a DFS, of migrating or replicating a directory (update activity and sharing are also important factors). If we ignore scans of the entire file system (at least 28 references per directory over the course of the week) we see that half of the directories on the system were not referenced at all (Figure 6-5 and Table 6-8). Many of the rest were referenced a few tens of times.

There were some net and system directories, though, that were referenced tens of thousands of times. Over half of the references, in fact, went to system directories referenced more than 100,000 times each. This is shown in Figure 6-6 and Table 6-9, where we have weighted the distributions in Figure 6-5 by the number of references made. This gives us the fraction of overall references as a function of directory activity. Note that 85% of the references went to directories referenced more than 10,000 times.

Figure 6-7 and Table 6-8 show, for each of the user cuts, the number of references made to *active* directories (those actually referenced given the cuts). If a directory was referenced at all by users (only 37% were), it was likely to see enough activity to make trying to minimize the access overhead (through caching, migration, or other mechanisms) worthwhile.

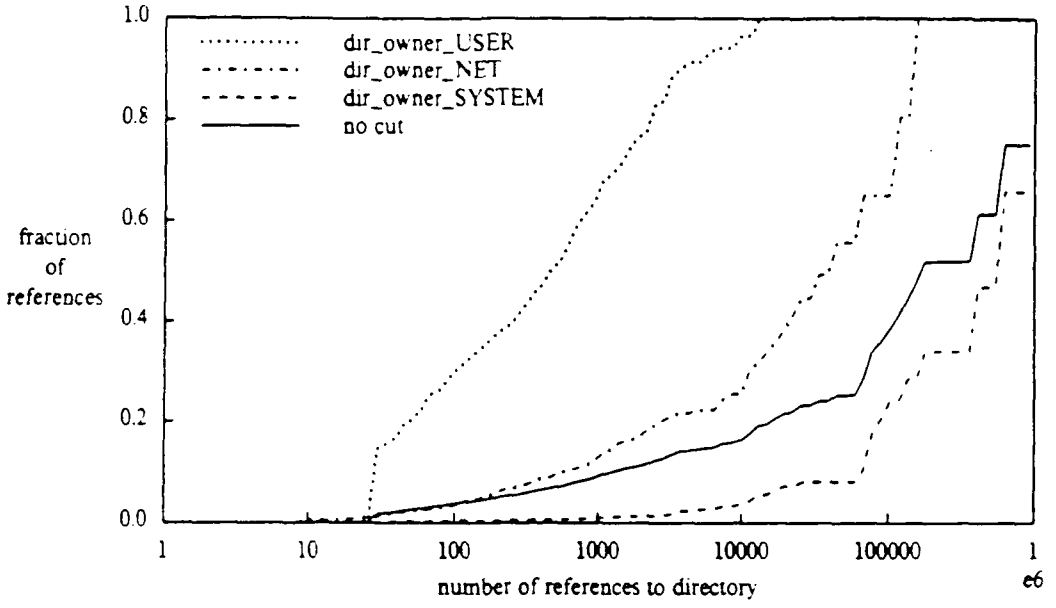


Figure 6-6: Fraction of references per directory (cumulative)

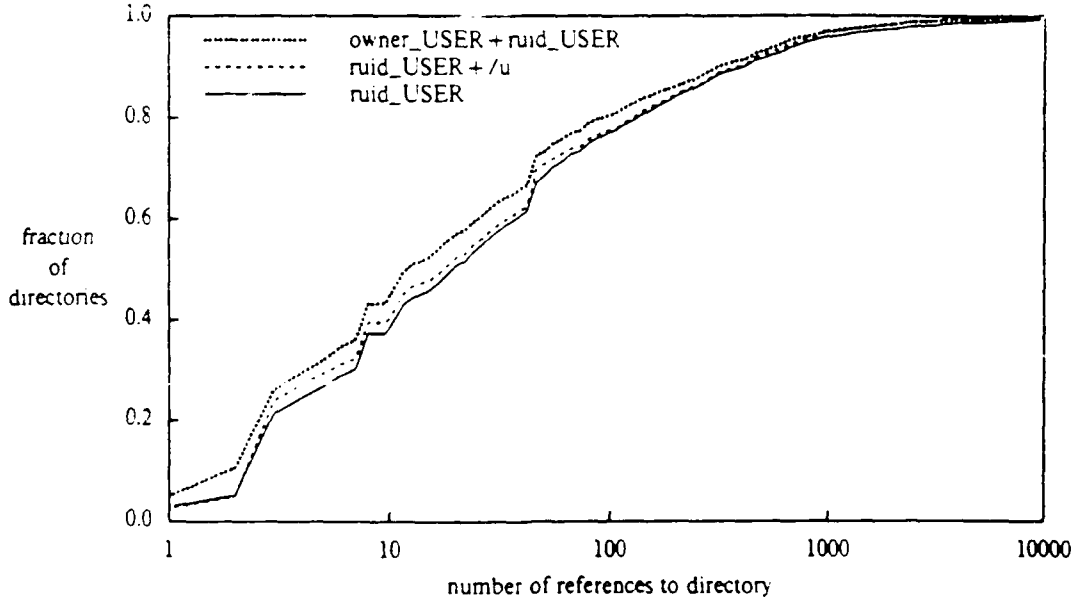


Figure 6-7: Number of references per active directory (cumulative, user cuts)

distribution	mean	median	std dev
dir_owner_NET	6.09e4	4.47e4	5.6e4
dir_owner_SYSTEM	5.54e5	6.17e5	4.1e5
dir_owner_USER	1.67e3	4.68e2	2.9e3
no cut	4.15e5	1.78e5	4.2e5

Table 6-9: Reference distribution (as a function of references/directory)

The most frequently referenced directories are listed in Table 6-10. Note that the four busiest directories accounted for over half of the references and received, between them, just 9 writes in a week. These directories are clearly very good candidates for extensive replication in a DFS, since update overhead is not an issue. The 15 most active directories accounted for 76% of the references. This suggests that even in a local environment, special treatment of a small number of directories could result in substantial improvements in name resolution performance.

The directories most frequently referenced by users in accessing their files and data are listed in Table 6-11. Most are shared user directories.

Knowledge of directory interreference intervals (the time from one reference of a directory to the next) is useful in estimating both the appropriate time scale for migration and the possibilities for caching. Figure 6-8 and Table 6-12 show that interreference intervals were short (opens to directories were strongly clustered). When a directory was referenced, the following reference (if any) had a 50% probability of occurring in the next 1/4 second. Part of this may be attributed to the heavily used system directories, but net and user directories also had short median interreference times[2]. There is strong reference locality in both time and space.

[2]The large fraction of zero length intervals for user directories was due to redundant references and opens to the current working directory, and to routines such as getwd that find the path of the current working directory by traversing up the directory tree to the root and then back down. That these are all binned at zero is partly an artifact of our analysis technique (see section 6.3.1).

references	fraction	reads	writes	reads/writes	path
1058380	25.0%	1058374	6	176400	/
587013	13.9%	587013	0	-	/usr
395132	9.3%	395129	3	131700	/usr/spool
168205	4.0%	168205	0	-	/usr/spool/rwho
145318	3.4%	118865	26453	4.49	/usr/spool/uucp
136363	3.2%	136121	242	562	/etc
114227	2.7%	104485	9742	10.7	/usr/spool/news
105239	2.5%	51857	53382	0.97	/tmp
85579	2.0%	85566	13	6580	/usr/lib
78696	1.9%	78696	0	-	/u
76778	1.8%	22820	53958	0.42	/usr/spool/mqueue
74590	1.8%	74590	0	-	/bin
71037	1.7%	71037	0	-	/dev
69093	1.6%	69093	0	-	/usr/spool/news/net
47737	1.1%	38770	8967	4.32	/usr/lib/news
35091	0.82%	35085	6	5800	/usr/spool/notes1.nyu

Table 6-10: Frequently referenced directories (no cut)

references	fraction	reads	writes	reads/writes	path
143956	23.4%	143954	2	72000	/
63235	10.3%	63235	0	-	/u
58955	9.6%	22917	36038	0.64	/tmp
27744	4.5%	27744	0	-	/usr
20988	3.4%	20988	0	-	/usr/spool
14836	2.4%	4614	10222	0.45	/usr/spool/mqueue
13138	2.1%	11770	1368	8.6	/u/ken
12286	2.0%	5521	6765	0.82	/usr/spool/mail
8967	1.5%	8963	4	2240	/u/ken/Src
6358	1.0%	4712	1646	2.86	/usr/spool/uucp
5980	0.97%	5336	644	8.29	/u/goddard/c400/assg2/coding
5328	0.86%	5233	95	55	/u/lee
5159	0.84%	4823	336	14.4	/usr/spool/news
3665	0.59%	3456	209	16.5	/u/scott/src/window
3510	0.57%	3510	0	-	/usr/local

Table 6-11: Frequently referenced directories (owner_USER+ruid_USER cut)

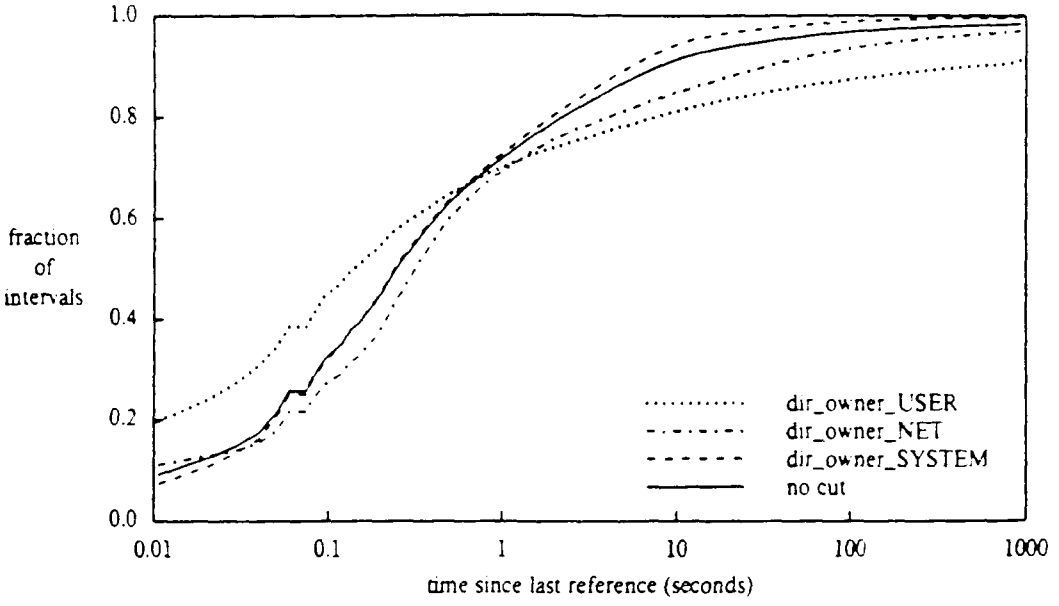


Figure 6-8: Directory inter-reference intervals (cumulative)

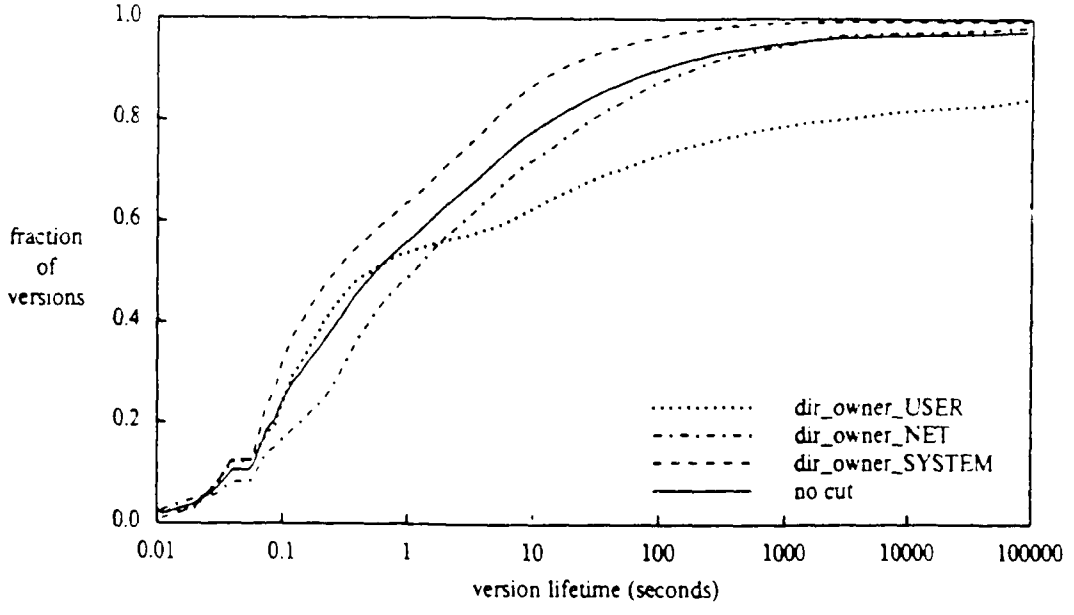


Figure 6-9: Directory version lifetimes (cumulative)

distribution	min	max	mean	median	std dev
dir_owner_NET	0	8.7e4	351	0.33	4.1e3
dir_owner_SYSTEM	0	8.7e4	73	0.27	2.2e3
dir_owner_USER	0	8.7e4	4680	0.15	1.8e4
no cut	0	8.7e4	560	0.27	6.4e3

Table 6-12: Directory inter-reference intervals (seconds)

Directory version lifetimes (the time from one write of a directory to the next) are shown in Figure 6-9. Versions whose lifetime extended beyond the logging period were given infinite lifetimes (lie to the right of the histogram). Note that half of all directory versions exist for a second or less. Part of the reason for these short version lifetimes is heavy write activity to the system directories /tmp and /usr/spool/mqueue and to net spool directories. Most system directories and the majority of user directories remained unchanged for relatively long periods of time.

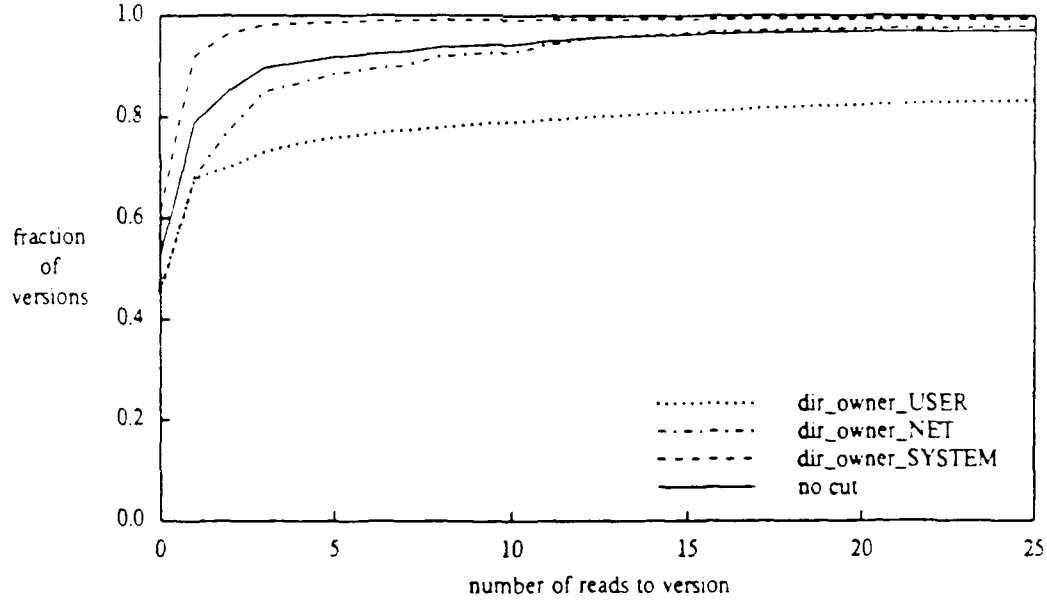


Figure 6-10: Reads per directory version (cumulative)

distribution	min	max	mean	median	std dev
dir_owner_NET	0	6.91e4	4.88	1	220
dir_owner_SYSTEM	0	6.17e5	23.9	0	2800
dir_owner_USER	0	3.90e3	14.2	1	74
owner_USER+ruid_USER	0	1.04e5	6.6	0	470
ruid_USER+/u	0	7.18e4	12.2	1	470
ruid_USER	0	2.38e5	11.7	0	970
no cut	0	6.17e5	14.3	0	1870

Table 6-13: Reads/directory version

Figure 6-10 and Table 6-13 present us with another view of directory versions: the number of reads that are made to any given version. Half of all versions are written again without being read. Roughly 4/5 of the remainder are read only a few times before being updated. While there are directory versions that can be safely cached or replicated regardless of the setup and update costs (some receive hundreds of thousands of references without being changed), separating them out from the majority of relatively useless versions may be difficult. Very cheap caching mechanisms, semantic knowledge, or knowledge of recent reference history would be useful here. For example, the knowledge that /tmp is used to store temporary files and so is frequently updated could be used to avoid potentially wasteful caching of this directory.

If we consider only user references to active directories holding user objects, we see similar distributions for reads per version (Figure 6-11 and Table 6-13). 86% of the versions received one or fewer user references before being updated by users. Directory versions on the /u file system saw slightly more read activity (not surprising, since heavily written system directories such as /tmp are not included here).

The first two columns of Table 6-14 show the mean number of readers per directory, as indicated by the account (ruid) of the reader, and the percentage of directories with more than *two* readers (we use two here because every directory is referenced by the housekeeping process). The next 4 columns show similar information for writers, but give the fraction with more than one writer, and for users (the overall number of distinct readers and writers). The last two columns show the

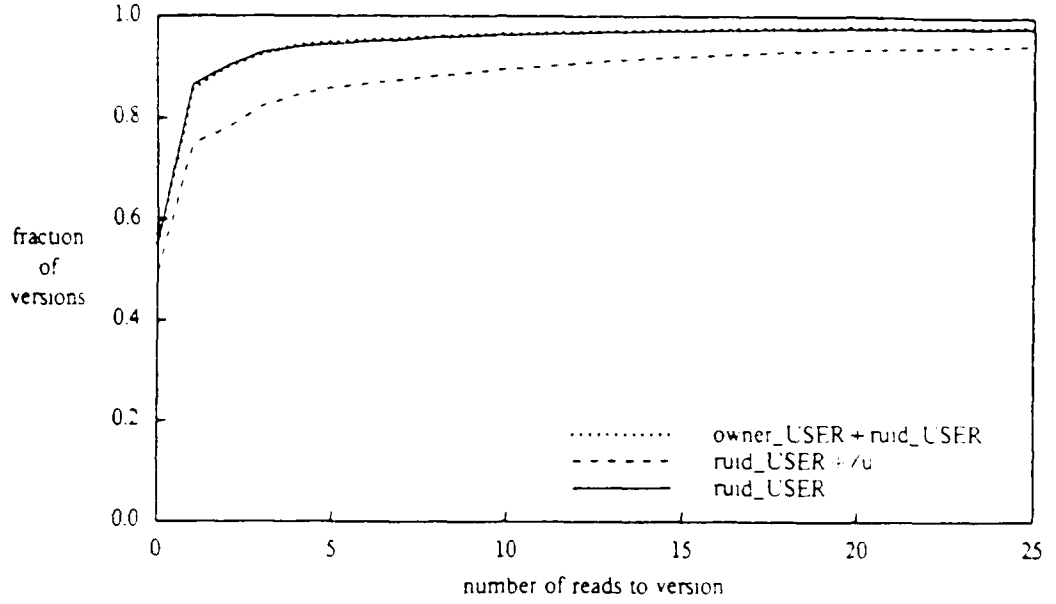


Figure 6-11: Reads per directory version (cumulative, user cuts)

cut	readers		writers		users (r ! w)		inversions	
	mean	>2	mean	>1	mean	>2	mean	max
dir_owner_NET	3.23	28.0%	1.35	14.6%	3.33	28.2%	34.6	6.56e3
dir_owner_SYSTEM	8.53	26.9%	0.857	3.0%	8.55	27.2%	1750	3.01e5
dir_owner_USER	1.60	9.0%	0.149	0.8%	1.60	9.0%	3.7	2.90e2
no cut	2.53	14.9%	0.487	4.2%	2.56	15.0%	149	3.01e5

Table 6-14: Directory sharing

cut	readers		writers		users (r ! w)		inversions	
	mean	>1	mean	>1	mean	>1	mean	max
owner_USER+ruid_USER	2.50	27.9%	0.609	1.3%	2.57	28.3%	41.2	2.9e4
ruid_USER+/u	1.47	18.8%	0.351	0%	1.49	18.8%	10.8	1.3e4

Table 6-15: Directory sharing (user cuts)

mean and maximum number of *inversions* per directory. The number of inversions is the number of times that the most recent user of the directory changes (this is basically the inversion clustering metric used by Porcar [Porcar 82]). For a directory used by only one user, the number of

inversions will be zero.

From Table 6-14 we can see that 15% of the directories had multiple users (users with separate accounts). Multiple readers were much more common than multiple writers. Most of the shared directories belonged to net and system accounts. These were predominantly directories containing news articles read by many users, spool directories accessed by a number of net accounts, and directories holding widely used system files. There was relatively little sharing of user directories. Shared system directories often had a number of active users and so a high number of inversions. In a distributed environment, replication or caching of these directories would be essential.

Statistics on the sharing of active directories holding user objects are given in Table 6-15. 1/5 of the active directories on the user file system were read by more than one user. None had multiple writers. Active directories used to resolve user objects showed a higher degree of sharing (because of shared system directories).

6.4.4. The High Cost of Opens

Based on the relatively small file sizes seen in studies of 4.2BSD systems and the long pathnames we have seen, it is reasonable to expect that directory overheads will be an important part of the cost of accessing a file. If we assume for the moment that no caching is done (or, for relative comparisons, that caching is equally effective for inodes, file data, and directory data), we can estimate both the number of disk blocks that are required to resolve a path and how this compares to the number of actual file data blocks that are read or written. Reading a UNIX directory requires reading a minimum of 2 blocks: one block containing the file descriptor (inode) for the directory and at least one data block. Assuming a block size of 512 bytes, directories with no holes (empty directory entries), an average of half the entries in a directory search for a name resolution, and no caching gives the distributions shown in Figure 6-12 and Table 6-16. The median of 7 blocks to resolve a path is impressively large, especially when compared to the median file size of 710 bytes

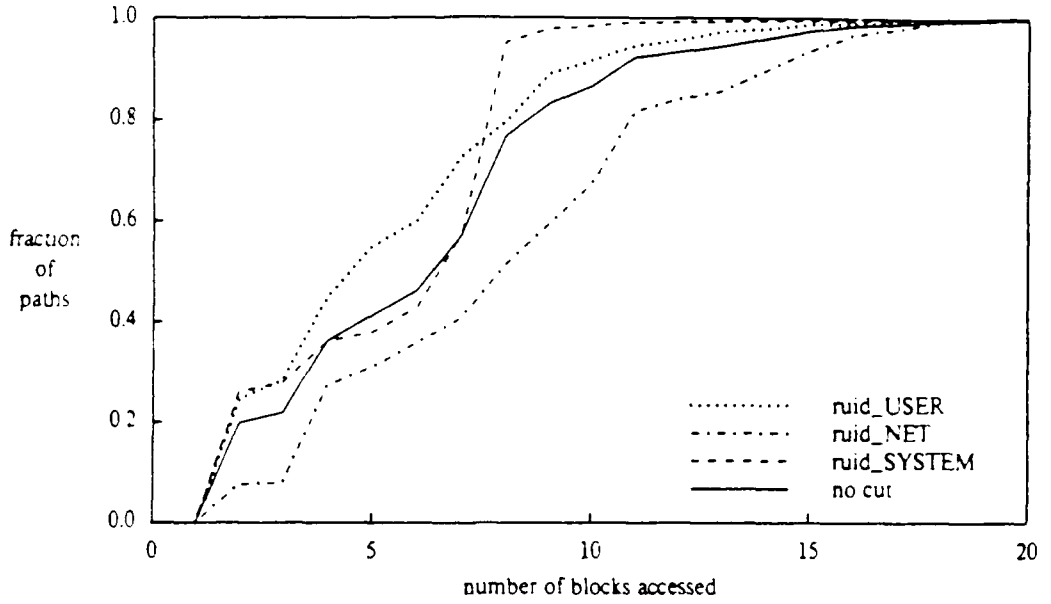


Figure 6-12: Path resolution cost (cumulative, 512 byte blocks)

distribution	min	max	mean	median	std dev
ruid_NET	2	22	8.43	8	4.2
ruid_SYSTEM	2	23	5.85	7	2.8
ruid_USER	2	28	5.74	5	3.4
no cut	2	28	6.61	7	3.7

Table 6-16: Blocks accessed/path resolution (512 byte blocks)

seen in our earlier study. Paths used by net processes are particularly long and hence expensive.

Accessing file data once a path is resolved also requires a minimum of 2 blocks: one block containing the file descriptor (we ignore indirect blocks here) and at least 1 data block. If we take the ratio of the blocks required for resolving an open path to the total number of blocks required (resolution cost plus file data cost based on the amount read or written and assuming contiguous access), we get the fraction of the cost (in blocks accessed) due to the directory overhead. This is shown in Figure 6-13 and Table 6-17.

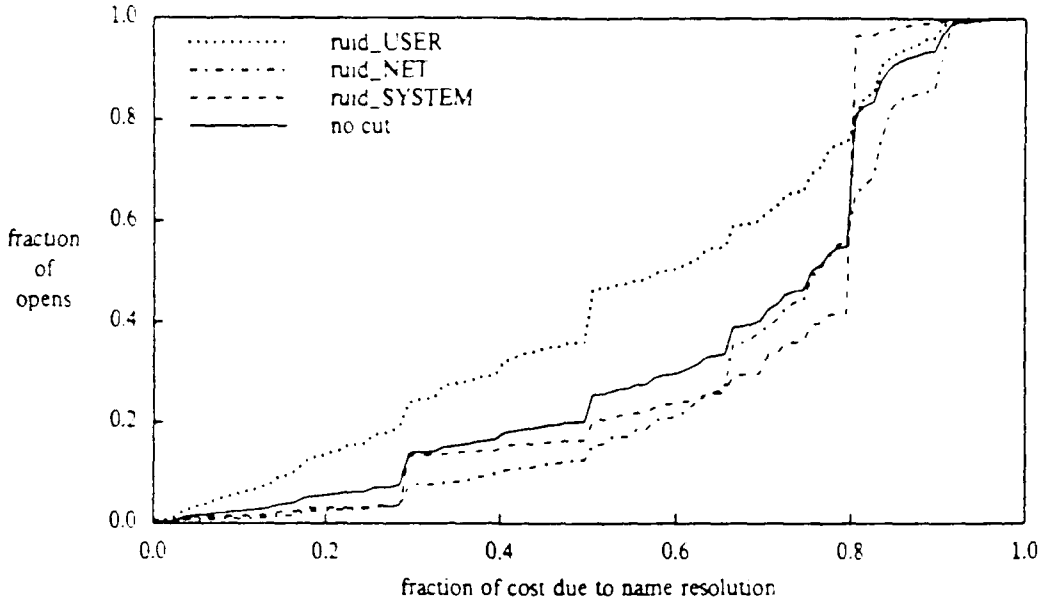


Figure 6-13: Name resolution overhead for file opens (cumulative, 512 byte blocks)

distribution	min	max	mean	median	std dev
ruid_NET	0.0005	0.99	0.71	0.77	0.19
ruid_SYSTEM	0.002	0.98	0.68	0.81	0.20
ruid_USER	0.0001	0.99	0.55	0.59	0.26
no cut	0.0001	0.99	0.66	0.76	0.22

Table 6-17: Directory overhead (512 byte blocks)

The directory overhead accounted for an average of 66% of the cost of accessing a regular file. This overhead accounted for the majority of the cost in 80% of the file accesses. For references made by user processes, the fraction of cost due to name resolution overhead is somewhat lower. This is due to users specifying shorter path lengths, accessing larger files, and reading a larger percentage of accessed files.

The cost distribution weights all files equally. This gives us useful information on the average overhead to access files (and so the effect of the overhead on response time), but is less useful in predicting the effect on throughput. For this we need the fraction of overall block requests that directory overhead accounts for. This information is given in Table 6-18. Note that half of all

type	NONE		ruid_NET		ruid_SYSTEM		ruid_USER	
	blocks	fraction	blocks	fraction	blocks	fraction	blocks	fraction
file data	6.32e6	46.9%	1.45e6	34.1%	1.56e6	38.3%	3.30e6	66.2%
file inode	7.54e5	5.6%	2.50e5	5.9%	2.98e5	7.3%	2.06e5	4.1%
directory data	4.01e6	29.8%	1.58e6	37.2%	1.26e6	31.0%	9.94e5	20.0%
directory inode	2.39e6	17.7%	9.65e5	22.8%	9.50e5	23.3%	4.83e5	9.7%
total	1.35e7	100%	4.23e6	100%	4.07e6	100%	4.98e6	100%

Table 6-18: Block counts for regular file opens, reads, and writes (512 byte blocks)

accesses were to directory data and inode blocks.

512 bytes is a small block size by today's standards. The 4.2BSD file system on Seneca uses a block size of 4096 bytes. Figures 6-14 and 6-15 and Tables 6-19 and 6-20 show what happens when we use the larger block size. The number of blocks required to resolve a path has dropped by 18%, but the fraction of cost due to the directory lookup overhead has risen sharply. It now makes up an average of 75% of the total cost and accounts for at least half of the cost in 97% of the file references. Big block sizes help most when reading file data. Directories and descriptor blocks are too small for the bigger block size to matter much.

Table 6-21 shows the no-caching breakdown of the number of blocks of various types accessed for a 4K byte maximum block size. Note that directory data and inode blocks now account for about 3/4 of the blocks accessed. The total number of blocks accessed has been reduced to 54% of the 512 byte maximum block size figures.

Figure 6-16 and Table 6-19 show the number of directory inode and data blocks accessed to resolve paths for our user cuts (assuming no caching and 4K byte maximum block sizes). Figure 6-17 and Table 6-20 give the corresponding cost distributions. For the ruid_USER+/u distribution, we have included all regular file references made by user processes, but only "charged" for blocks on the /u file system. References to files on other file systems have zero resolution cost

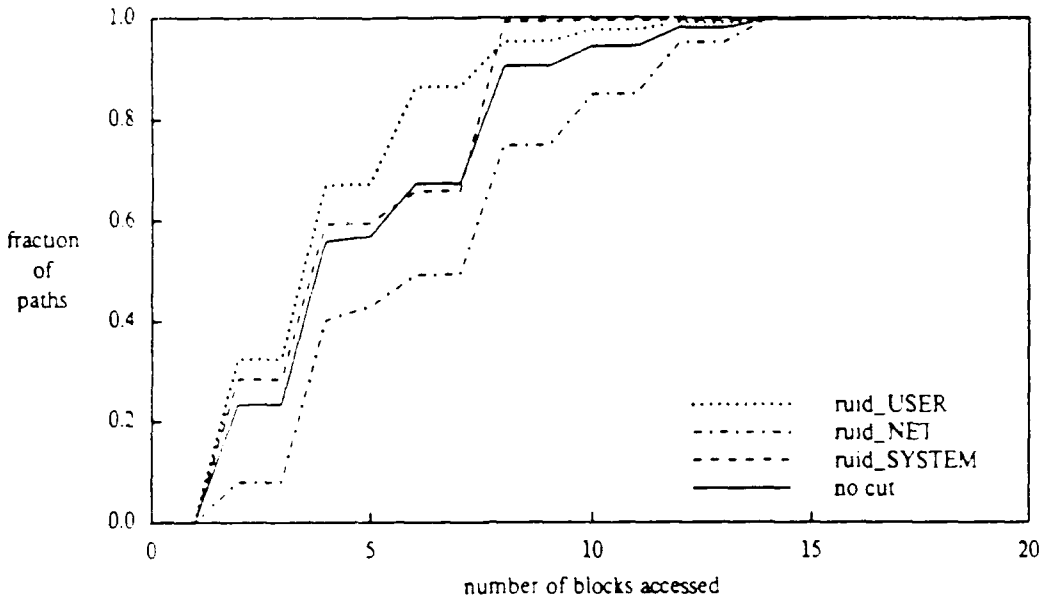


Figure 6-14: Path resolution cost (cumulative, 4K byte blocks)

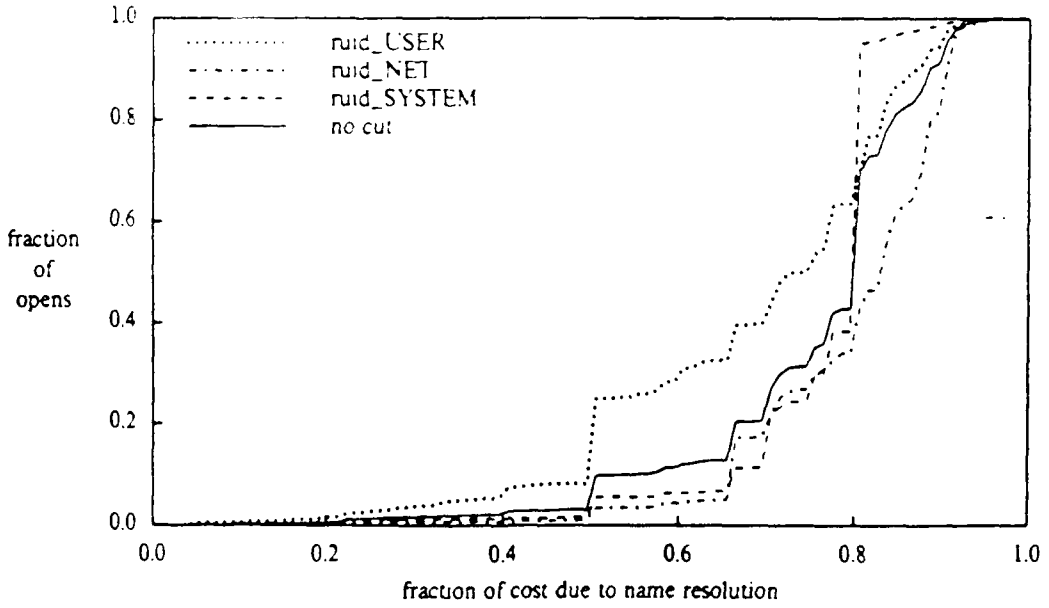


Figure 6-15: Name resolution overhead for file opens (cumulative, 4K byte blocks)

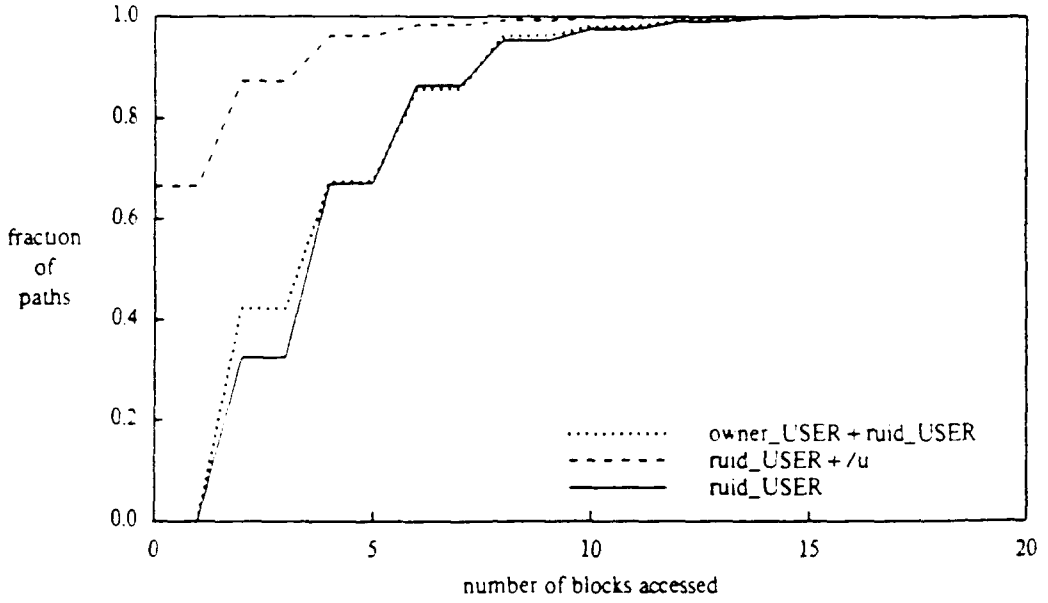


Figure 6-16: Path resolution cost (cumulative, 4K byte blocks, user cuts)

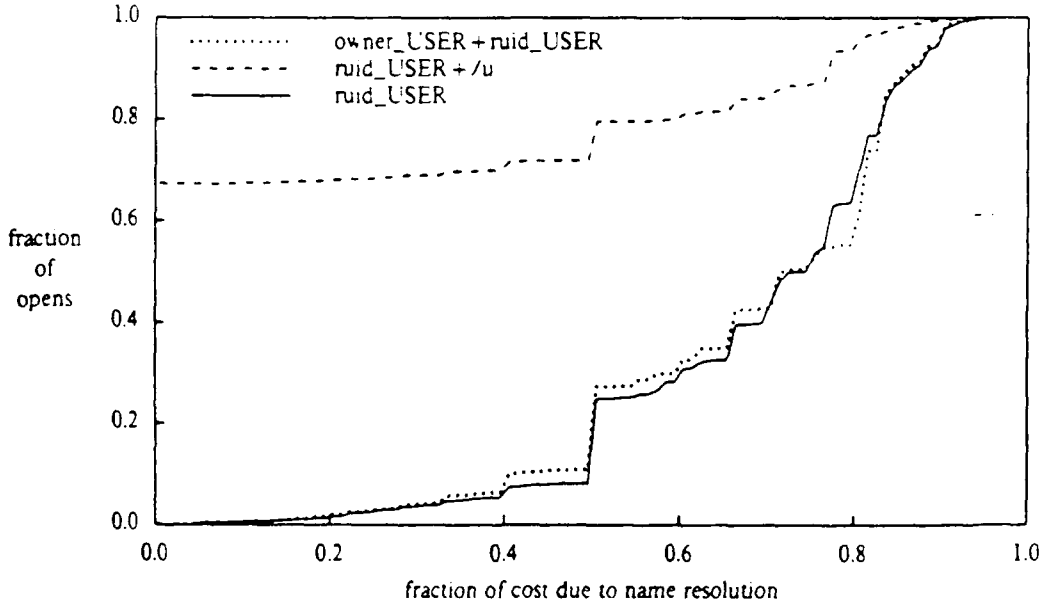


Figure 6-17: Name resolution overhead for file opens (cumulative, 4K byte blocks, user cuts)

distribution	min	max	mean	median	std dev
ruid_NET	2	16	6.93	8	3.3
ruid_SYSTEM	2	16	4.97	4	2.5
ruid_USER	2	22	4.44	4	2.5
owner_USER+ruid_USER	2	22	4.22	4	2.5
ruid_USER+/u	0	18	1.05	0	1.8
no cut	2	22	5.40	4	3.0

Table 6-19: Blocks accessed/path resolution (4K byte blocks)

distribution	min	max	mean	median	std dev
ruid_NET	0.004	0.99	0.80	0.84	0.11
ruid_SYSTEM	0.01	0.98	0.76	0.81	0.10
ruid_USER	0.001	0.99	0.69	0.74	0.18
owner_USER+ruid_USER	0.001	0.98	0.68	0.73	0.18
ruid_USER+/u	0	0.99	0.21	0	0.32
no cut	0.001	0.99	0.75	0.81	0.14

Table 6-20: Directory overhead (4K byte blocks)

type	NONE		ruid_NET		ruid_SYSTEM		ruid_USER	
	blocks	fraction	blocks	fraction	blocks	fraction	blocks	fraction
file data	1.26e6	17.3%	3.23e5	12.1%	4.07e5	15.4%	5.27e5	28.5%
file inode	7.54e5	10.3%	2.50e5	9.4%	2.98e5	11.3%	2.06e5	11.1%
directory data	2.90e6	39.7%	1.12e6	42.2%	9.94e5	37.5%	6.36e5	34.3%
directory inode	2.39e6	32.7%	9.65e5	36.3%	9.50e5	35.9%	4.83e5	26.1%
total	7.30e6	100%	2.66e6	100%	2.6536	100%	1.85e6	100%

Table 6-21: Block counts for regular file opens, reads, and writes (4K byte blocks)

here and the cost of resolving in "/" for absolute paths is not included. Since 65% of the files referenced by user processes were actually on other file systems, the average directory overhead for this cut is low.

It should be noted that the results in this section don't apply directly to BSD UNIX. 4.2BSD maintains an extensive cache of inode, directory, and file data. 4.3BSD has added a cache of

recently used directory entries. These results show, though, that "hidden costs" in UNIX file systems are significant and demonstrate how rapidly their importance increases as the block size increases.

6.5. Summary

This chapter has analyzed in some detail directory reference patterns resulting from primarily open activity on a 4.2BSD UNIX system supporting university research. As with the results of our file reference studies, the results should be applied to other situations with care. The major findings of our analysis:

(1) Directories are mostly small, with half holding under 8 entries. Referenced directories are somewhat larger (median of 26 entries).

(2) 3/4 of all references are made to system directories. Most references go to a few very active system directories.

(3) Reads account for 93.4% of the references we see and writes for 6.6% of the references.

(4) 70% of all paths are specified absolutely. Relatively few paths (26%) reference objects in current working directories.

(5) Paths are "long", having an average of 2.70 components.

(6) Interreference times are short. Half are 1/4 second or less.

(7) Directory versions are usually short lived (half live less than a second) and receive few reads.

(8) The combination of small file sizes and long access paths means that name resolution overhead is high. In the absence of caching and using a 4K byte maximum block size, 72% of the blocks accessed in opening and using files are for name resolution.

The implications of these results for DFS design will be explored in Chapter 7.

Chapter 7

Implications for the Design of Distributed File Systems

7.1. Introduction

File and directory reference patterns can have a substantial effect on file system behavior. This effect ranges from catastrophic congestion and failure when file systems are used in ways that designers never anticipated or allowed for, to enhanced availability and performance when file systems are able to anticipate and adjust to requests. Careful file system design and dynamic "tuning" are particularly important in a distributed file system. In a DFS, network overhead, scaling and congestion issues, the potential for parallelism, independent failure modes, and greater flexibility in design make both the penalty for failure and the rewards for success much more dramatic.

Chapters 5 and 6 concentrated on studying short term file and directory reference patterns in UNIX environments. In this chapter we use the results of those studies to investigate mechanisms used by Roe and to evaluate their strengths and weaknesses. We also briefly examine implications the data have for the design and behavior of distributed file systems in general.

We start, in section 7.2, by examining the availability that one can expect in a Roe system that sees reference patterns similar to what we have measured. Section 7.3 considers the implications our measurements have for file placement and migration algorithms. Section 7.4 examines performance issues raised by our measurements and section 7.5 briefly summarizes the chapter.

It should be emphasized that the observations and analysis presented here are most applicable to systems that see reference patterns similar to ours. They will not necessarily carry over to other environments.

7.2. Availability

7.2.1. Availability Model

In Chapter 2 we defined availability to be the fraction of valid requests that are successful, and assumed that all request failures are due to hardware or catastrophic operating system failures. In the discussion that follows we represent host availability by a single number, the availability averaged over the time period of interest. This figure ranges from 0 for a permanently inaccessible host to 1.0 in the unlikely case that a host is always accessible. Machines on the University of Rochester Computer Science Department network typically have an average availability of roughly 0.98. This is equivalent to one half hour crash per day, 2 days of down time every 100 days, or some combination of these. An average availability figure does not completely represent host failure modes. It does, however, allow us to produce analytic results that will give us a good feeling for the issues involved. We will return to this point in section 7.2.3.

For a distributed file system, the availability for a given call depends on the availability of the name service and the availability of the object being referenced. If we take the case of Roe, which supports a hierarchical directory distributed at the node level, the name service availability is in turn dependent on the availability of the path components. Assume that these availabilities are independent. This implies that all required path components and the target object are on distinct hosts, and that these hosts fail independently. The availability for a given call is then just the

product of the availabilities of each of the path components and of the target. Let a_i be the availability of the ith path component, a_{target} the availability of the object being referenced by the path (if any), and n the path length. The overall availability for a given call is then

$$a_{call} = \left[\prod_{i=1}^{n} a_i\right] a_{target}$$

We have, from Chapter 6, the distribution of path lengths we can expect. Letting p_n be the fraction of paths seen that have n components, we can express the overall availability of the system as follows.

$$a_{overall} = \sum_{n=1}^{\infty} p_n a_{call} = \sum_{n=1}^{\infty} p_n \left[\prod_{i=1}^{n} a_i\right] a_{target}$$

The availability of an unreplicated component is the host availability, a. If a resource is replicated, the availability may be found by calculating the probability of each state of the resource and then summing over the states that result in the resource being available. If weighted voting is being used and all hosts have equal availabilities, the read availability of a c copy suite having one vote per copy and a read quorum r is given by [Smith 84]:

$$a_{WV} = \sum_{j=r}^{c} \frac{c!}{j!(c-j)!} a^j (1-a)^{c-j}$$

The write availability is found in a similar fashion.

Another replication method that has been suggested for use in local area network environments is *available copies* [Bernstein 84]. With this algorithm, an operation succeeds if even one copy of a resource is available. This is appropriate when consistency is not important or when an unavailable copy is known to be unavailable everywhere and that it can be updated before being used. The availability of available copies when c copies are used is given by

$$a_{AC} = 1-(1-a)^c$$

7.2.2. Basic Distributed File System Availability

Using the results given by these equations and the distribution of path lengths we found, we can calculate the availability of a variety of distributed file systems. Figure 7-1 and Table 7-1 show the results for a central file server (located entirely on one host), an unreplicated but distributed file system, one replicated using available copies (3 copies of each object), and a system replicated using weighted voting (3 copies, with a read quorum of 2 and a write quorum of 2). For weighted voting, the triple "(2,2,3)" specifies the read quorum, the write quorum, and the total number of copies, in that order.

In this figure, "worst case" refers to our assumption that all path components and the target of the call are on distinct hosts (that is, failures are independent). "Best case" is the situation where all objects referenced in a call are on the same host, or n hosts in the case of objects replicated n times (components on a particular path do *not* fail independently). We include both the worst and

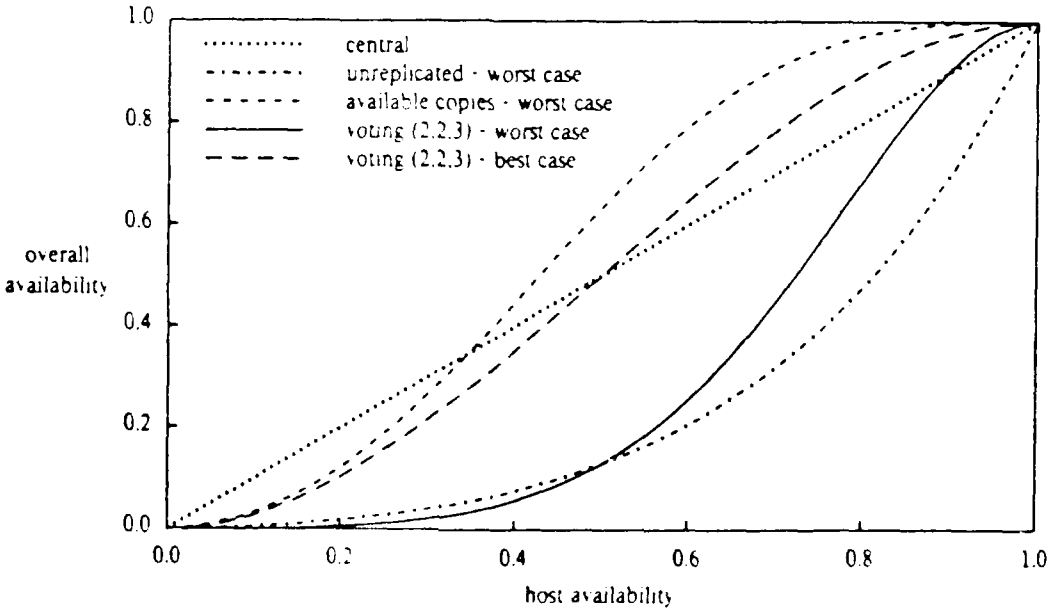


Figure 7-1: Availability based on path length distributions

host availability	central server	unreplicated	available copies	voting (2,2,3) worst case	voting (2,2,3) best case
0.6	0.6	0.21	0.79	0.26	0.65
0.8	0.8	0.48	0.972	0.69	0.89
0.9	0.9	0.7	0.996	0.904	0.972
0.95	0.95	0.84	0.9996	0.974	0.9928
0.98	0.98	0.931	0.99997	0.9958	0.9988
0.99	0.99	0.965	>0.99999	0.9989	0.9997

Table 7-1: Availability based on path length distributions

best cases for our weighted voting configuration. The best case here is clearly considerably better than the worst case. This shows the benefits to be had by grouping together related resources. In Roe this can be done by highly replicating root directories and by migration algorithms that tend to favor grouping together objects that are used together. The flexible approach used by Roe also allows the system to place new resources to avoid hosts that are down or that have relatively low availability (this is not represented in the model we are using here). Other DFSs such as NFS [Lyon 85] and LOCUS [Walker 83b] are distributed by subtree. NFS is unreplicated, which makes its availability at best equivalent to the central server approach. LOCUS doesn't attempt to preserve availability, and so its availability is closer to that of available copies. The approach used by Roe allows for more flexibility in adapting to changing needs and networks but, as we see here, care must be taken to preserve availability.

Available copies fares well compared to the rest of the algorithms used here. This is a classic example of trading off consistency (or limiting the domain) for availability. Available copies is, in fact, highly available, but in an environment where partitions can occur or updates can be temporarily "lost" due to host failures, it can allow old versions of files to reappear. Because of this it violates our transparency and user model requirements.

If we look in more detail at availabilities we expect to see during normal operation (Figure 7-2), we see that all of the replication methods considered here are able to significantly enhance availability. It should be clear from this that transparent access to replicated resources can be used to

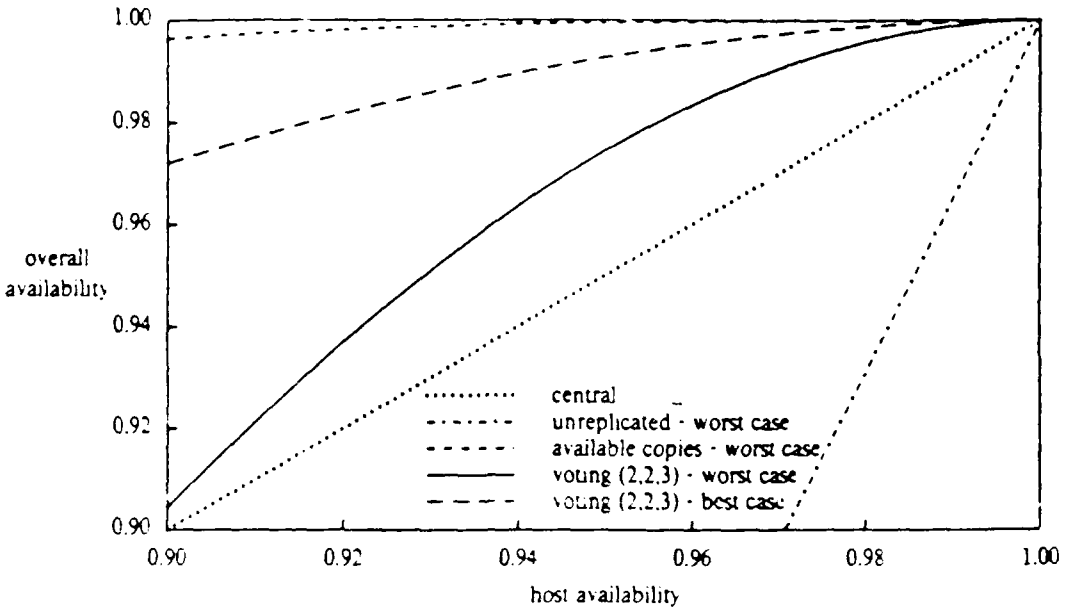


Figure 7-2: Availability based on path length distributions, a > 0.9

support file systems with higher availability than could be expected in a centralized system, regardless of the distribution algorithm used.

7.2.3. Adding Read/Write and Semantic Information

One of the attractive features of weighted voting is the ability to adjust read and write quorums to take advantage of differing read and write rates. We saw in our data that 93.4% of directory accesses and slightly over half of file accesses were for read. Using 5 copies of each object, with a read quorum of 2 and a write quorum of 4, gives the results shown in Figures 7-3 and 7-4 and in Table 7-2. Again, these are all worst case results (no grouping of related resources). These figures show the read and write availability of the system separately. The write availability is about the same as what we saw with the (2,2,3) configuration. Even though the availability of an object actually being written is lower, the availability of directory components read to reach the object is considerably higher, leading to no degradation. The read availability is dramatically better. Taken together, the result is a significant improvement in overall availability without any increased access overhead for reads, which are by far the most frequent operation on Roe objects.

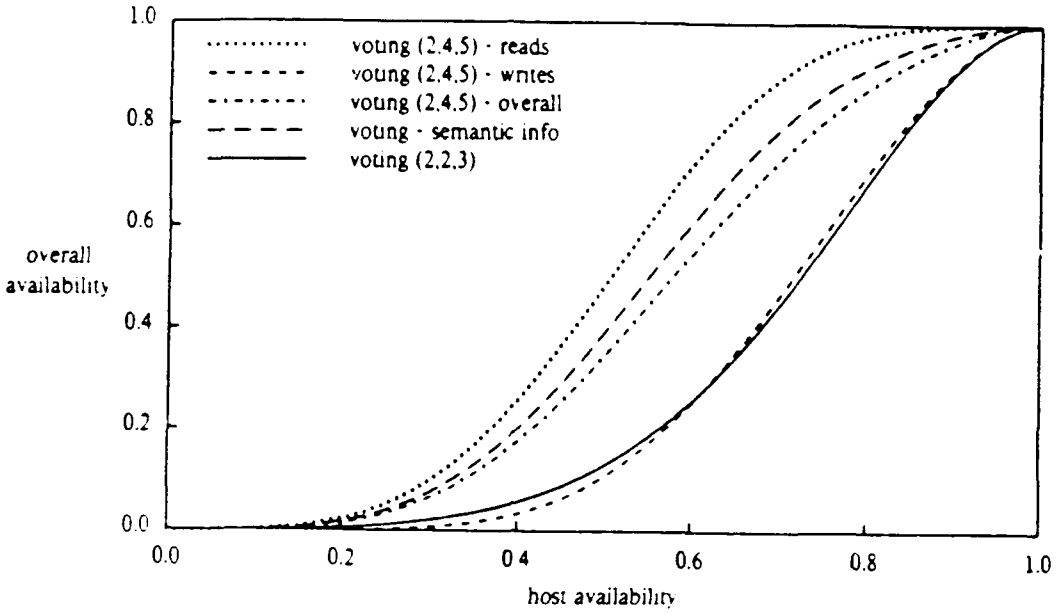


Figure 7-3: Worst case availability using semantic and read/write information

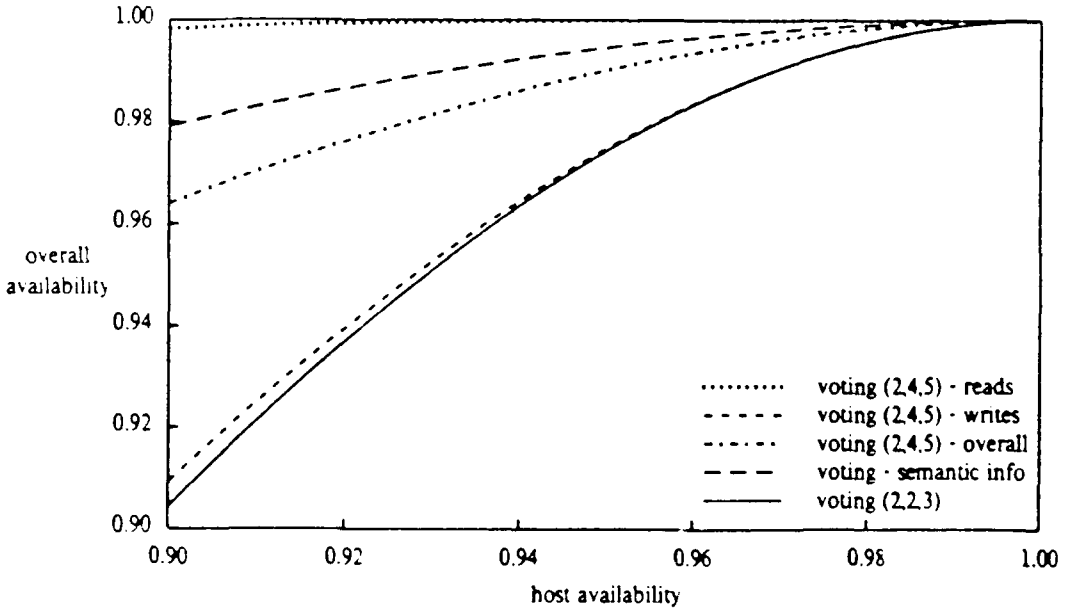


Figure 7-4: Worst case availability using semantic and read/write information, a > 0.9

host availability	voting (2,4,5) reads	voting (2,4,5) writes	voting (2,4,5) overall	voting semantic info
0.6	0.72	0.25	0.55	0.61
0.8	0.975	0.70	0.87	0.91
0.9	0.9983	0.909	0.964	0.979
0.95	0.9999	0.975	0.9902	0.9947
0.98	>0.99999	0.9957	0.9983	0.9991
0.99	>0.99999	0.9989	0.9996	0.9998

Table 7-2: Worst case availability using semantic and read/write information

More improvements may be had by using semantic information to decide on a replication factor. We know, for example, that temp and log files have a lower read/write ratio than perm files. There are also several directories that we identified in our studies as having particularly low read/write ratios. If we use a voting configuration of (2,2,3) for these files and directories, while retaining the (2,4,5) configuration for everything else, we arrive at the curve shown in Figures 7-3 and 7-4 (we have assumed here that path length distributions for requests with differing configurations are the same). Once again, we see a significant improvement. Based on this, it is clear that replication strategies that make use of semantic information or past usage history can significantly increase the availability of a system. At our reference host availability of 0.98, even this fairly simple use of semantic and read/write information decreases the probability of an operation failing by 80% when compared to voting without using this information. The probability of an operation failing is now less than 5% of what it would be in the central server (NFS-like) case.

We can further increase availability by making more extensive use of read/write and semantic information. For example, some directories and files in our study were heavily read and rarely, if ever, written. Giving them a voting configuration of (1,3,3) makes their availability equivalent to what we saw for available copies.

We have assumed up to this point that hosts on a network have uniform availabilities. This is unlikely to be true in a heterogeneous environment. The Roe network model includes estimates of host availabilities, and the placement algorithm that we described in Chapter 3 places resources

based on these estimates. Further, hosts generally fail, are inaccessible for some period of time, and then recover. Roe recognizes these failures and reconfigures around them. These considerations, combined with the short file lifetimes that we measured and the use of placement and migration algorithms that group related resources, mean that the calculations we have made will tend to underestimate the actual availability of a running Roe system. More accurate availability figures may be acquired using a simulation of Roe or measurements of the running system, but the results we have presented above provide conclusive evidence of Roe's ability to provide highly available files.

7.3. Reconfigurability: Initial Placement and Migration

Our analysis of file and directory reference patterns in Chapters 5 and 6 revealed striking differences in reference patterns between classes of files. Differences also exist in the reference patterns generated by each of the user classes. The large differences between classes point out a need for a distributed file system, such as Roe, that can adapt to and exploit these differences. In this section we examine placement and migration policy considerations based on file class, user class, and other characteristics of the data we have observed.

The differing reference patterns between file classes can be used to help determine appropriate initial placements that are based in the intended use of the file. For example, most temp files in our environment were opened only once or twice. These files were also short lived, generally existing for only a few seconds. There is no need to replicate a temp or other short-lived file. In addition, the very short lifetime of these files argues for placing based on delay and other performance factors, not on availability. In contrast to temp files, the overwhelming majority of perm files had lifetimes that extended beyond the week long logging period. Migration can be expected to be an issue for these files.

We also saw differences based on the owner and user of files. Data in user perm files were long lived (Figure 5-34), which was not the case for the overall data. The long life of data in these files indicates that they will benefit from placement that favors reading over writing.

The differing placement needs of the various file types also points out the need for a consistency control algorithm that can be adjusted to take these needs into account. As we saw in section 7.2, our use of weighted voting, which lets us specify placement quorums based on anticipated read/write ratios, allows us to use these differences to increase availability without a significant effect on performance. This issue is particularly important for directories, where our data showed that a few directories accounted for the majority of reads and were rarely modified. User perm files also had high read/write ratios.

While files of a given type tended to occur more frequently in some directories than others, there were many cases where perm, log, and temp files all occurred in a single directory. Because of this, the common practice of distributing and configuring directory subtrees and the files that reside in them based on expected usage is not always appropriate. Significant differences exist between files even within a given subtree. For a DFS such as Roe, information from the user on the type of a file or on its expected usage will allow for more effective placement.

Three figures that are useful in estimating the appropriateness of dynamic migration are the size of a file, the fraction of a file opened for reading that is actually read, and the fraction of a file opened for writing that is actually written. Most referenced files in our environment were small, with the median file size being 710 bytes. Log files were significantly larger, with a median size of 39,000 bytes. Referenced directories were also small. We found that 68% of files opened with read access were completely read and 78% of files opened with write access were completely written. The percentage read and written depended strongly on the class of the file (log, perm, or temp), the mode of open, the file opener, and the size of the file. In particular, log files are almost never completely written, users completely read 94% of files they open read-only, and large files are rarely completely accessed.

The high percentage of a file that was read or written, combined with the small file sizes we have observed, tells us that migrating a file as a whole, the approach used by Roe, is usually appropri-

ate. Log and very large files are an exception[1]. In environments limited by bandwidth considerations, file class, open mode, user, and file size will provide a simple basis for making migration decisions.

Knowledge of file interopen and directory interreference intervals are useful in estimating both the appropriate time scale for migration and the possibilities for caching. The short intervals that we saw for files (a median of 60 seconds) and directories (a median of just 0.27 seconds) suggests that fast response to changing patterns is important. User files had substantially longer interopen intervals, making quick migration a less critical issue in their case.

The number of opens by a user to a file gives an indication of the potential benefits of migrating the file to a user's machine. Locality of reference and the degree of sharing are also factors here. Most files in our environment were opened only once or twice. However, most opens went to files opened many times. 75% went to files opened more than 10 times and half to files opened more than 480 times. For these frequently opened files (and hence for a distributed file system as a whole), migration, caching, and replication may be useful. This will be especially true for perm files, since they receive so much open activity per file and tend to be opened read-only. As we mentioned above, file sharing is also a factor here. Overall, only about 12% of the files on the system were shared. Few users files were shared. Other classes of files saw more sharing, but, except for heavily used system and news log, perm and executable files, sharing was incidental to the normal use of the file. Most sharing was read-only, indicating that extensive replication of these files is appropriate.

The lack of file sharing and the locality of reference implied by the short interopen times we see for both files and directories suggest that migration algorithms that respond to references by migrating a copy to the user's host or by caching a copy, subject to the file size and log constraints we mentioned, are appropriate. For most files it will not be necessary to worry about "migratory thrashing" or frequent cache invalidation due to accesses from other users. This is

[1]One might prefer to use a different logging mechanism in a distributed environment in any case.

particularly the case with user files, which are effectively never shared. The lack of sharing also provides justification for our earlier decision to lock on a whole file basis in Roe.

The bursty nature of requests (for our background activity in particular) indicates that congestion may be a serious problem at times. Results from the VICE/Andrew system [Svobodova 85] confirm the importance of this issue. Algorithms that place and migrate files to minimize congestion will be particularly appropriate in the case of background activity of this nature.

7.4. Performance

7.4.1. General Considerations

Our results may also be used to investigate performance issues in distributed file systems, and as an aid in pointing out areas where problems are likely to occur. In this section we summarize results that are relevant to DFS performance and briefly describe their significance. The following two sections examine the performance implications of these results for Roe, and for distributed file systems in general.

We found that referenced files in our environment were small, with half being less than 710 bytes long, and 75% less than 4096 bytes long (a common block transfer size). We also found that path lengths tended to be relatively long, with an average of almost 3 name components per path. Taken together, these two results suggest that directory lookup and open overhead will tend to dominate file access time, particularly in a distributed environment.

Directories were generally small, with half holding under 8 entries. Referenced directories were somewhat larger, but also relatively small, with a median size of 26 entries per directory. This means that the trend towards larger block sizes won't help name resolution costs and will, in fact, work to increase their relative importance. However, the small size of directories does mean that there is little space overhead for caching them.

70% of the logged paths were absolute references. This implies that deep directory trees raise the cost of references. The root of the file system must be cheap to access, since it will be heavily used.

93.4% of directory references were for read in our data (the actual overall figure for the system was probably somewhat higher). Clearly directories should generally be optimized for lookup. Some directories are heavily written and rarely read, though. Other organizations or placements may be appropriate for these directories. The use of semantic information or past history would be useful here.

Reads accounted for 84% of the bytes transferred in the system. Many of these reads were from large administrative files that were frequently read and rarely if ever written. Replication and caching of even a few such files could substantially increase the performance of a DFS.

There was an extremely high degree of locality of reference to files and directories. Half of all opens to perm files went to just 0.7% of the files. Over half of all execute requests went to just 13 files. The 15 most active directories accounted for 76% of the references. This intensely localized activity suggests that even a very modest amount of caching or other special treatment for such files and directories could produce significant improvements in system performance.

We found that most directory versions were short lived. Over half lasted less than a second. This was due largely to high update rates to system and net scratch and spool directories. There is little point in writing these versions back to disk. Delaying writes of such directories could be expected to improve performance (although not, perhaps, reliability).

Most directory versions also received only a few references. Half of all versions were written again without being read. Roughly 4/5 of the remaining versions were read only a few times before being updated. This, combined with the short lifetime of most versions, implies that caching these versions serves no purpose. However, some directories were heavily read and rarely, if ever, updated. The four busiest directories accounted for over half of the references and received, between them, just 9 writes in a week. Versions of these directories could be safely cached or

replicated regardless of the setup and update costs. Separating them out from the majority of relatively useless versions may be difficult, though. Very cheap caching mechanisms, semantic knowledge, or knowledge of recent reference history would be useful here. For example, the knowledge that /tmp is used to store temporary files and so is frequently updated could be used to avoid potentially wasteful caching of this directory.

Directory activity was concentrated on system and network directories. References to user directories accounted for less than 10% of the overall directory references in the system. There was little sharing of user directories.

As we described in section 7.3, we saw substantial differences in reference patterns between the file and user classes we examined. Placement and migration decisions made based on these differences can be used to improve the overall performance of a system. There is, for example, generally no need to replicate a temp file, and files (such as user perm files) that are likely to have high read/write ratios can be replicated and placed to favor reads. Significant differences between classes were seen in areas such as reference locality, number of opens over the logging period, read/write ratios, data lifetime, and interopen intervals. These class differences are described in section 7.3 and in Chapters 5 and 6.

7.4.2. Performance Issues in Roe

The small file sizes and relatively long paths that we observed indicate that the overhead to open a file in Roe will be a significant component of the access cost. We saw indications of this in the UNIX name resolution overhead studies that we performed in Chapter 6. Roe separates directories and the objects that they reference, and allows them to migrate independently. The open overhead imposed by this separation and distribution can be high relative to a more integrated approach, which makes this cost of special concern in Roe.

The cost to open a file can be broken down into two components: 1) name resolution cost, and 2) the cost to actually open the (possibly replicated) file once it is found. Roe addresses name resolution cost by caching directory information. The cost to perform the actual open can be reduced in

Roe using the advisory locks that we alluded to in Chapters 3 and 4, or by adjusting voting quorums based on expected usage.

Our measurements of high directory reference locality, combined with the small size of directories and the low level of updates to many heavily used directories, suggest that a directory cache can be an effective, low cost method for reducing name resolution overhead. To test this idea, we simulated an LRU whole directory cache using our reference trace as input. We found (Figure 7-5 and Table 7-3) that even a cache holding as few as 10 directory nodes achieved an 85% hit ratio. A 30 node cache gave a 95% hit ratio.

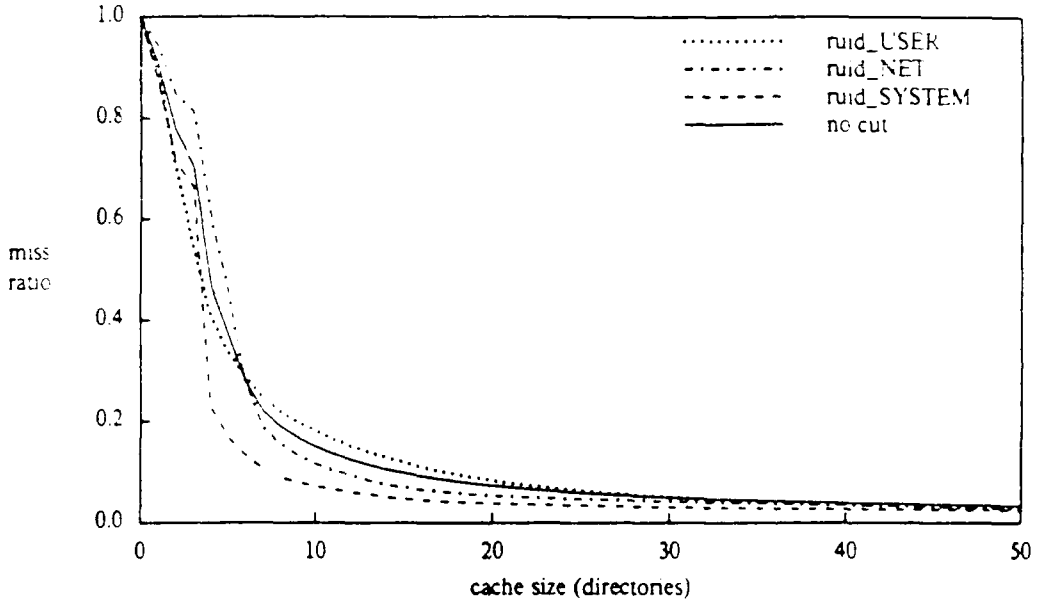


Figure 7-5: Whole directory cache effectiveness

nodes	no cut	ruid_NET	ruid_SYSTEM	ruid_USER	owner_USER+ ruid_USER	ruid_USER+ /u
5	0.37	0.46	0.17	0.34	0.25	0.14
10	0.15	0.12	0.072	0.18	0.11	0.079
15	0.099	0.070	0.045	0.12	0.076	0.065
30	0.050	0.042	0.031	0.050	0.047	0.051

Table 7-3: Miss ratio vs. cache size (in nodes)

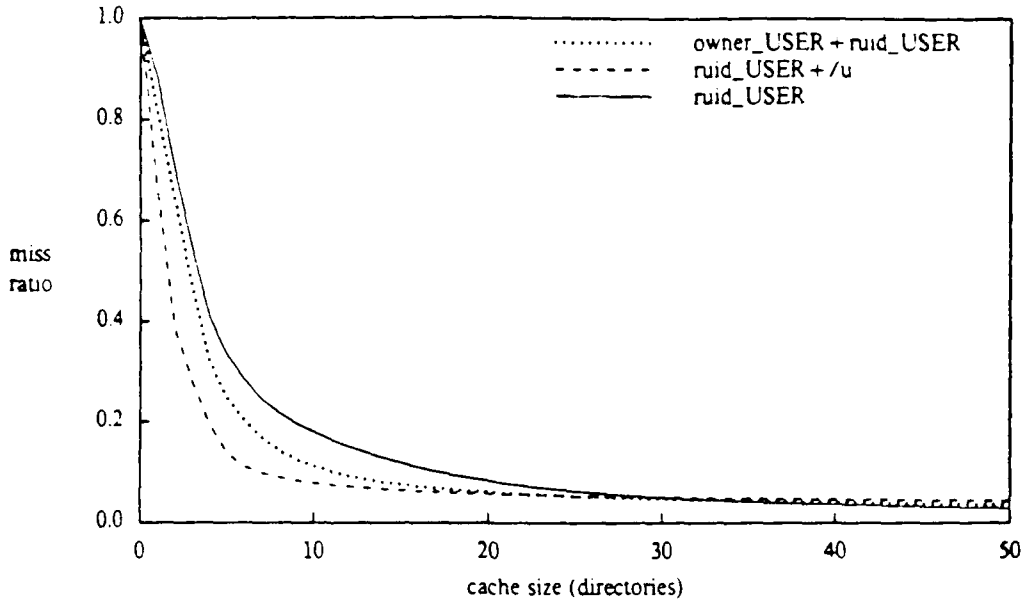


Figure 7-6: Whole directory cache effectiveness (user cuts)

bytes	no cut	ruid_NET	ruid_SYSTEM	ruid_USER
5K	0.37	0.40	0.22	0.40
10K	0.20	0.16	0.11	0.24
15K	0.14	0.10	0.067	0.17
20K	0.10	0.079	0.052	0.13
30K	0.070	0.060	0.036	0.075
40K	0.052	0.043	0.030	0.047

Table 7-4: Miss ratio vs. cache size (in bytes)

User references to user objects also show a high degree of locality (Figure 7-6 and Table 7-3). A 10 node LRU whole directory cache captured 92% of user references to the user file system (/u) and 89% of references required to reach user objects.

Since directories are not generally very big, whole directory caches don't require much space (Figure 7-7 and Table 7-4). For the overall trace, a 14K byte cache gave the 85% hit ratio seen with the 10 node cache. Using a 41K byte cache raised the hit ratio to 95%.

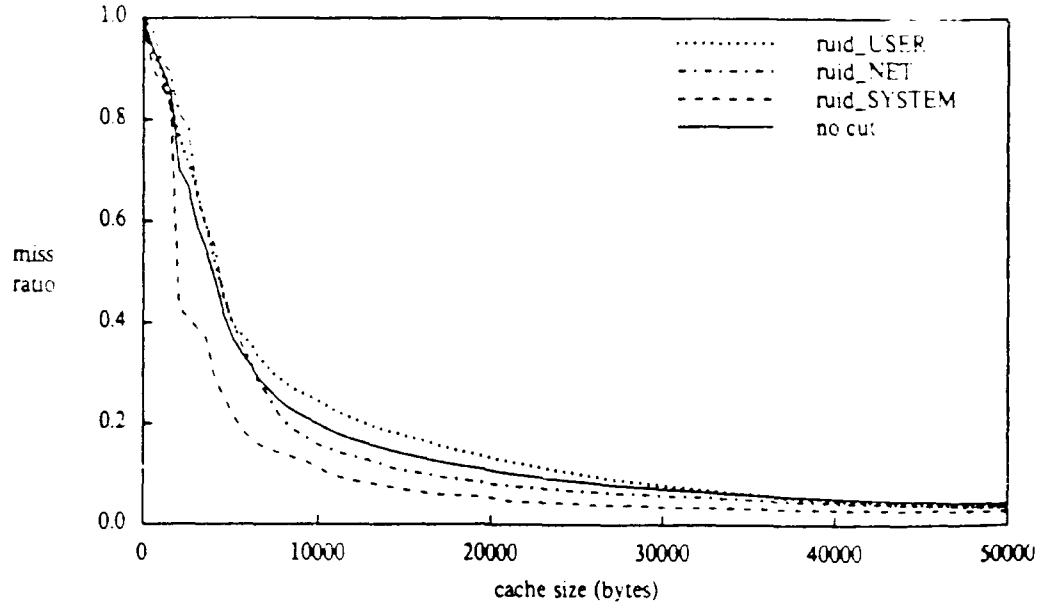


Figure 7-7: Whole directory cache effectiveness (byte size limit)

As we mentioned in Chapter 6, our log of file system activity doesn't include all directory references. Adding in the *lstat* and *stat* calls missing from our data could be expected to increase the effectiveness of our cache. These calls usually follow an open of the directory referencing the object of the status call and so they will all result in "hits" on the cache. Using the estimates of *lstat* and *stat* frequency made in section 6.3 and assuming short paths leads to a 15%-20% decrease in the miss ratios given above.

Two other studies of directory caching in UNIX environments focussed on page level caching [Sheltzer 86] and entry level caching [Leffler 84, Leffler 86]. Sheltzer et al. looked at page level caching for references on a LOCUS system [Walker 83b] (an enhanced, distributed version of 4.1BSD UNIX). Their simulations assume, though, that directories fit in a single page. While this is true for most directories on a BSD UNIX system, there are a few large, heavily used system directories that typically contain in excess of 100 entries (see figure 6-3). These directories are referenced frequently enough to make the high hit ratios found by Sheltzer et al. questionable for page sizes one might typically use with directories (1K bytes or less because of the small size of directories). As Figure 6-3 shows, inferring dynamic distributions from static ones can be

dangerous. Their simulations are actually, then, for a whole directory LRU cache. Their result of hit ratios of 87%-96% (depending on the system) for a 40 page (node) cache agrees with our result of a 96% hit ratio for a 40 node cache and indicates that our results are typical for the environment.

Leffler et al., in tuning and enhancing 4.2BSD, found that a system-wide entry level cache containing 400 entries (about 18K bytes) gave a 60% hit ratio. This was coupled with a per process directory offset cache having a 25% hit ratio (to catch directory scans), giving an overall hit ratio of 85%. This is effectively an entry cache and a per process single directory cache, with invalidation on update. We saw a hit ratio of 89% for an 18K byte whole directory cache. Our hit ratios are higher because we effectively "read ahead" for processes scanning directories by caching the entire node on 1st reference, don't invalidate on update or working directory changes, and cache globally.

Two other factors affect directory caching in a distributed environment: 1) the amount of directory sharing, and 2) the update rate for shared nodes. Directories that are widely cached will be relatively expensive to update, due to the need to send out update notifications to each cached copy. As we indicated in the previous section, the busiest directories (those near the root of the file system) are heavily read and rarely if ever written. These may be cached with impunity. Most other system directories were also rarely modified, and so are appropriate candidates for distributed caching.

Sharing of user directories is incidental to their normal use. For these directories, caching is not really necessary. Migration to the current location of the user is a more appropriate approach.

The outlook is less bright for shared, frequently updated system and net directories such as /tmp and spool directories. As we saw earlier, most versions of these directories were short-lived and received few if any reads. Maintaining a coherent distributed cache of such directories would be prohibitively expensive. Roe could recognize these directories based on usage histories or semantic information, and respond by not caching them and by optimizing them for writes. An

alternative, used by LOCUS and Andrew, is to remove such directories from the global name space, making them visible only from the local machine. However, this reduces transparency, making it difficult to distribute applications that use such directories, and making it impossible to balance load across servers and local machines. Other possible approaches here are attempting to cache at the entry level, partitioning the name space of these directories in some fashion (perhaps by user), or recognizing the inherently undistributed nature of such directories and eliminating them.

Opening a replicated file once the name is resolved generally involves contacting multiple copies of the file and collecting quorum information. One technique we have suggested is maintaining *advisory locks* on frequently opened files that are broken, with notification, if the file is opened for writing. This is similar to the "broken read lock" mechanism that Gifford proposed for Violet [Gifford 79a], although they were proposed there as a means of increasing concurrency.

This approach would work best if files are generally not shared or, for files that are shared, usually accessed read-only. In addition, a high degree of reference locality is necessary to minimize the overhead involved in setting up and maintaining locks. As we have seen, these are in fact the conditions that we see for most files in the system. The primary exceptions are temp files, which would generally not be created as replicated files, and log files, which were frequently written and rarely read.

For files whose references are highly localized in time, an alternative to advisory locks is collapsing the quorum down to a single or a few copies when the file is first accessed, and then restoring the original quorum when the burst of accesses has finished. This trades a small loss in availability for performance. The high degree of locality we have seen suggests that this approach has promise, but further analysis would be necessary to understand the issues involved.

7.4.3. Implications for Other Distributed File System Approaches

Name resolution and open overheads will also be an important consideration in other DFS designs. As we saw in Chapter 6, the combination of relatively long path names and small files means that,

in the absence of caching, the majority of UNIX file system activity is in support of name resolution. The same considerations will hold for DFS designs that interpret remote directories locally. Optimizations in the area of name resolution are likely to produce significant performance improvements, particularly as the block size used for data transfers increases.

One approach to dealing with this problem, taken by the V system [Cheriton 84], is to group names and objects together, and to have servers holding objects manage name interpretation themselves. This reduces distribution possibilities (and hence transparency), but it also reduces the cost of resolving names for remote objects.

For designs that access remote directories locally, caching or some other means of short circuiting remote operations will be essential. We found that entry level caches, by themselves, are much less effective than node level caches because of the frequent sequential access of entries in a directory. Caching strategies that recognize and exploit this sequentiality will do well.

Large block sizes help when reading some files, but many files, most directories and all descriptor blocks are too small to benefit from larger block sizes. For file systems in general (and so for DFS file servers), coupling larger block sizes with a file system design that ties file descriptors, directories, and data together on disk could be expected to lead to substantial performance improvements. The 4.2BSD file system does this by attempting to allocate these related items close together on disk [McKusick 84]. Further improvement (at the cost of increased crash recovery complexity) could be achieved by allocating inodes and at least the initial data in files contiguously on disk. On a 4K block size file system, this could be expected to lead to about a 40% decrease in transfers for file open, read, and write activity, given equivalent caching effectiveness for data, inodes and directories. This observation is similar to one made by Mullender and Tanenbaum [Mullender 84], although our results are based on an analysis of actual file and directory reference patterns (as opposed to a less reliable static analysis of file sizes).

Nearly 3/4 of all references went to system directories. Less than 10% went to user directories. Further, only about 15% of file opens were to files owned by users, and some of these were temp

files placed in system scratch directories. For DFSs, such as Andrew [Satyanarayanan 85], that support access to local file systems coupled with access to a global user file system, minimizing the performance impact of adding the global file system on local accesses is clearly important. Conversely, carefully coupling transparent access to a network file system holding user files with cheap access to local files can result in a coherent distributed file system with good overall performance, even in situations where network or server access is expensive.

A significant drawback of this approach is that a user's file requests are concentrated on a single host (the local one). In the case of Andrew, where a primary goal was to support large numbers of clients with relatively few server and network resources, this is appropriate. However, the potential performance benefits of balancing load across multiple servers and of the parallelism possible with distributed access are lost when file systems are segregated in this fashion.

7.5. Summary

This chapter has examined some of the implications the patterns we saw in our file and directory reference data have for distributed file systems in general, and for Roe in particular.

One potential area of concern in the Roe design was that the node-level fragmentation and distribution of directories would seriously impact the overall availability of Roe. We found that with the reference patterns we saw and using a simple probabilistic failure model, Roe can use replication to provide significantly increased availability over a centralized server, even assuming worst-case distribution of components. Further, the flexibility inherent in the Roe replication approach allows it to make use of read/write rates, semantic information and failure information to further increase availability without impacting the performance for reads, the most frequent operation.

An examination of the reference data characteristics affecting migration showed that Roe's approach to migrating files and directories as a whole is the correct one in our environment. The strong locality of reference we see for files and directories dictates a short time scale for migration. We saw substantial differences in access patterns for various classes of files and users that

can be used in making placement and migration decisions.

Further examination of the data pointed to name resolution and file open costs as potential performance concerns in Roe. The high degree of locality in directory references led us to investigate whole node LRU directory caching as a possible solution. We found that, because of the locality, even a small cache gave good results. This approach appears workable for most directories. However, there are a small number of frequently updated system directories that do not appear to be cacheable in a distributed environment. Finally, we examined advisory file locks as a way to reduce the open overhead for replicated files. Highly localized file references combined with a low write rate for most shared files make advisory locks an attractive option.

Chapter 8

Summary and Future Work

8.1. Summary

The major claim of this dissertation is embodied in our thesis statement:

> Full network transparency in distributed file systems offers significant benefits. These benefits include increased availability, more effective use of resources, the ability to adapt to changing demands, transparent reconfiguration to adjust to changes in resources, a greatly simplified file system model from the user/application point of view and, with careful design, enhanced performance over other distributed file system approaches.

As we pointed out in Chapter 1, evaluating the validity of this thesis had been hampered by two factors:

(1) Limited experience with the design and implementation of transparent distributed file systems.

(2) A lack of understanding of the ways in which file systems are used.

We have addressed the first shortcoming by designing and implementing Roe, a fully transparent distributed file system. Roe supports a substantially higher degree of transparency than earlier distributed file systems, and is able to do this in a heterogeneous environment. Roe provides a coherent framework for uniting techniques in the areas of naming, replication, consistency control, file and directory placement, and file and directory migration. Roe does this in a way that provides full network transparency. This network transparency allows Roe to provide increased availability, automatic reconfiguration, effective use of resources, a simplified file system model, and important performance benefits.

We have addressed the second problem by collecting data on file and directory references from a large UNIX system. Our analysis of these data provides by far the most detailed information to date on short-term file reference patterns in the UNIX environment. In addition to examining the overall request behavior, it breaks references down by the type of file, owner of file, and type of user. We find significant differences in reference patterns between the various classes. These differences emphasize the need for a distributed file system, such as Roe, that can adapt to and exploit them.

Our study also provides, for the first time, information on directory reference patterns in a hierarchical file system. The results provide striking evidence for the importance of name resolution overhead in UNIX environments and supply information necessary to design algorithms that minimize this overhead, both in single site and distributed file systems.

Two potential concerns in Roe are the availability impact of its distributed structure and its performance. We have used the results of our reference studies to investigate these issues. We find that Roe's ability to transparently replicate resources based on semantic and usage information allows it to provide enhanced availability over other commonly used approaches. File open overhead proves to be a performance issue in Roe, but simple techniques exist for reducing its impact.

Taken together, the results we have presented in this dissertation are both a compelling justification for our thesis statement and a significant contribution to Computer Science.

8.2. Future Work

The work described in this dissertation is not the final word on network transparency, or on distributed file systems. It provides a foundation for future research in many directions. This section briefly describes some possibilities for future research based on the work we have presented.

8.2.1. Extension and Evaluation of Roe

The Roe design that we have presented here doesn't explicitly address security issues. It would be straightforward to add file and directory level security mechanisms (such as access control lists) given our assumption of a single administrative domain. However, if Roe crosses administrative domains or is on a network that includes untrusted hosts, these mechanisms will not be adequate. Untrusted hosts can be incorporated, with some loss of performance, by performing validations outside of these hosts and not placing protected files and directories on them. Validation in the presence of multiple administrative domains is perhaps best done in the context of the domain where the object was created.

Multiple administrative domains also raise the issue of autonomy. In an environment containing multiple domains it will generally not be appropriate for Roe to place resources created in one domain in another domain, or to have operation in one domain be dependent on operation in another. A more appropriate model in this environment may be that of mutually suspicious Roe systems interacting to provide a global service. Representing this in Roe and understanding the effect that it would have on transparency requires further work.

Roe was designed for high bandwidth, low delay networks. Its ability to migrate and place resources based on usage and network considerations may make it useful in network environments where bandwidth and delay are important considerations. Further work is needed to understand the impact of our design decisions on operation in such environments.

The directory algorithms that we presented in Chapter 3 were intended to support low overhead caching of directory nodes that had relatively low update rates, and so provide concurrency control

and voting at the node level. Not all directories in the data we saw fit this model. It would be interesting to investigate the possibility of algorithms that support quick verification of currency at the node level but allow concurrency control to be invoked at the entry level. An alternative might be to support entry-level caching for frequently updated directories.

The design of Roe is conceptually fairly simple. However, the flexibility inherent in the approach used by Roe makes predicting the behavior difficult. In Chapter 7 we did a simple analysis of Roe availability and performance based on the data we collected. A more detailed analysis, based on simulation or on actual usage, is necessary to gain a greater understanding of Roe.

Measurements that would help evaluate the Roe design include the overhead for directory accesses, file opens, replication, consistency control, and state maintenance, the read/write performance under various replication configurations, and the availability of the system. Two additional measurements that are particularly sensitive to work load are overall resource utilization and the distribution of network and disk traffic devoted to various activities.

A trace driven simulation using the data we have collected as input is one way to gain experience with Roe and to collect this information. There is, however, no substitute for using a system in the environment for which it was intended. Using Roe on a day-to-day basis would provide experience and feedback that no simulation could match.

8.2.2. Reference Data Collection

The collection and analysis of file system traces can soak up endless resources. We stopped at the point were we felt that we had enough information to understand the implications that our data had for file system design. There is a great deal of additional collection and analysis that could be done. Some possibilities include:

- Studies of open frequency as a function of file age. Smith found that for long term file reference patterns, open frequency falls off as the age of the file increases [Smith 81a]. A survey of files on Seneca made at about the time our data were collected showed that 2/3 of all user files (user log and perm files) hadn't been accessed in over one month

[Friedberg 85]. This suggests that Smith's finding holds true in our environment for at least some classes of files.

- Studies of interopen intervals as a function of file size. Porcar found that smaller files tend to have shorter interopen intervals [Porcar 82]. We don't expect this to be true for the overall activity in our system (because of the large heavily used administrative files), but it may be true for user files.
- Measuring the paging and inode access activity. It would be interesting to see what fraction of the file system bandwidth is devoted to each of these activities.
- Examining in more detail the activity per user. This information would be useful in designing distributed file systems that include personal workstations. Ousterhout et al. [Ousterhout 85] have done some of this work.
- Fitting curves to various distributions (size, interopen time, and so on). These would be useful in writing synthetic drivers for use in simulating distributed file systems [Satyanarayanan 83].
- Investigating the correlation between directory depth and activity. Since most paths are absolute, one would expect that directories close to the root of the directory tree will be referenced more frequently than those towards the leaves.
- Further data collection and analysis for different environments and work loads. This would give us a better feeling for where our data fit into the universe of file system usage.

8.2.3. Initial Placement and Migration Algorithms

Entire dissertations have been written on the subject of file placement and migration algorithms. This might have been one of them, had we not found the architectural and transparency issues involved in supporting automatic placement and migration so interesting.

Roe provides architectural support for a wide range of placement and migration algorithms. The file and directory reference traces that we have collected provide a rich source of data for use in

designing and evaluating algorithms. Possibilities for research in the context of Roe include:

- Placement and migration based on the class of the file and user. The substantial differences we saw in reference patterns between the various classes should make them useful predictors of future usage.
- The use of additional parameters specified by file creators to place and later migrate files.
- The amount of information that should be kept in a network model to allow placement and migration to be done effectively.
- The importance of the currency of network model information.
- The interaction between initial placement and migration algorithms.
- Placement and migration algorithms that dynamically balance server load to reduce congestion.
- Algorithms that selectively attempt to increase performance or availability.
- Algorithms that dynamically vary the number of copies of a resource. Porcar has done some initial studies of on-demand replication [Porcar 82] that show this to be a promising approach.
- The effect of migration on network and host resource utilization in Roe.
- Algorithms that act based on usage information collected by Roe. This could include both migration algorithms that make use of information in individual files, and placement algorithms that act based on overall user behavior.
- Algorithms that attempt to group files and directories together. Two possible uses of grouping information are prefetching to increase performance and co-locating resources that are used together to increase effective availability.

Bibliography

[Archibald 84] Archibald, J. and Baer, J., "An Economical Solution to the Cache Coherence Problem," *Proceedings of the 11th International Symposium on Computer Architecture*, 1984, 355-362.

[Ball 76] Ball, J., Feldman, J., Low, J., Rashid, R. and Rovner, P., "RIG, Rochester's Intelligent Gateway: System Overview," *IEEE Transactions on Software Engineering* 2:4, December 1976, 321-328.

[Bannister 82] Bannister, J. and Trivedi, K., "Task and File Allocation in Fault-Tolerant Distributed Systems," *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, July 1982, 103-111.

[Barbara 86] Barbara, D. and Garcia-Molina, H., "The Vulnerability of Vote Assignment," *ACM Transactions on Computer Systems* 4:3, August 1986, 187-213.

[Bernstein 82] Bernstein, P. and Goodman, N., "A Sophisticate's Introduction to Distributed Database Concurrency Control," TR-19-82, Aiken Computer Laboratory, Harvard University, 1982.

[Bernstein 84] Bernstein, P. A. and Goodman, N., "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Transactions on Database Systems* 9:4, December 1984, 596-615.

[Birrell 80] Birrell, A. and Needham, R., "A Universal File Server," *IEEE Transactions on Software Engineering SE-6*:5, September 1980, 450-453.

[Birrell 82] Birrell, A., Levin, R., Needham, R. and Schroeder, M., "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM* 25:4, April 1982, 260-274.

[Birrell 84] Birrell, A. and Nelson, B., "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems* 2:1, February 1984, 39-59.

[Black 85] Black, A., Lazowska, E., Levy, H., Notkin, D., Sanislo, J. and Zahorjan, J., "An Approach to Accommodating Heterogeneity," TR 85-10-04, Department of Computer Science, University of Washington, October 1985.

[Bloch 87] Bloch, J., Daniels, D. and Spector, A., "A Weighted Voting Algorithm for Replicated Directories," *Journal of the ACM 34*:4, October 1987, 859-909.

[Boggs 80] Boggs, D., Shoch, J., Taft, E. and Metcalfe, R., "Pup: An Internetwork Architecture," *IEEE Transactions on Communications COM-28*:4, April 1980, 612-623.

[Bukys 82] Bukys, L. and Floyd, R., "Even More FTP Cogitation," Internal Document, Computer Science Department, University of Rochester, June 1982.

[Bukys 83] Bukys, L., Private communication, April 1983.

[Cheriton 83] Cheriton, D. and Zwaenepoel, W., "The Distributed V Kernel and its Performance for Diskless Workstations," *Operating Systems Review 17*:5, October 1983, 129-140.

[Cheriton 84] Cheriton, D. and Mann, T., "Uniform Access to Distributed Name Interpretation in the V-System," *Proceedings of the 4th International Conference on Distributed Computing Systems*, May 1984, 290-297.

[Cooper 85] Cooper, E., "Replicated Distributed Programs," *Operating Systems Review 19*:5, December 1985, 63-78.

[Davcev 85] Davcev, D. and Burkhard, W., "Consistency and Recovery Control for Replicated Files," *Operating Systems Review 19*:5, December 1985, 87-96.

[Dean 87] Dean, M., Sands, R. and Schantz, R., "Canonical Data Representation in the Cronus Distributed Operating System," *Proceedings of the IEEE Infocom '87*, March 1987, 814-819.

[Dolev 82] Dolev, D. and Strong, R., "Distributed Commit with Bounded Waiting," IBM Research Report RJ3417, San Jose, CA, March 1982.

[Dowdy 82] Dowdy, L. and Foster, D., "Comparative Models of the File Assignment Problem," *ACM Computing Surveys 14*:2, June 1982, 287-313.

[Dugan 86] Dugan, J. and Ciardo, G., "Stochastic Petri Net Analysis of a Replicated File System," TR CS-1987-1, Department of Computer Science, Duke University, December 1986.

[Floyd 85] Floyd, R. A., "Short Term File Reference Patterns in a UNIX Environment: Preliminary Results," Internal Note, Department of Computer Science, University of Rochester, August 1985.

[Fowler 85] Fowler, R., "Decentralized Object Finding Using Forwarding Addresses," TR 85-12-01, Department of Computer Science, University of Washington, December 1985.

[Fridrich 81] Fridrich, M. and Older, W., "The Felix File Server," *Operating Systems Review 15*:5, December 1981, 37-44.

[Fridrich 84] Fridrich, M. and Older, W., "HELIX: The Architecture of a Distributed File System," *Proceedings of the 4th International Conference on Distributed Computing Systems*, May 1984, 422-431.

[Friedberg 85] Friedberg, S., Private communication, July 1985.

[Gait 86] Gait, J., "Highly Available, Enhanced Response File Service in Network Computers," *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986, 548-554.

[Garcia-Molina 82] Garcia-Molina, H., "Elections in a Distributed Computing System," *IEEE Transactions on Computers C-31*:1, January 1982, 48-59.

[Garcia-Molina 84] Garcia-Molina, H. and Barbara, D., "Optimizing the Reliability Provided by Voting Mechanisms," *Proceedings of the 4th International Conference on Distributed Computing Systems*, May 1984, 340-346.

[Garey 79] Garey, M. and Johnson, D., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[Gifford 79a] Gifford, D., "Violet, An Experimental Decentralized System," TR CSL-79-12, Xerox Palo Alto Research Center, September 1979.

[Gifford 79b] Gifford, D., "Weighted Voting for Replicated Data," *Operating Systems Review 13*:5, December 1979, 150-163.

[Gifford 82] Gifford, D., "Information Storage in a Decentralized Computer System," TR CSL-81-8, Xerox, March 1982.

[Gifford 88] Gifford, D., Needham, R. and Schroeder, M., "The Cedar File System," *Communications of the ACM 31*:3, March 1988, 288-298.

[Goldberg 83] Goldberg, A., Popek, G. and Lavenberg, S., "A Validated Distributed System Performance Model," *Proceedings 9th International Symposium on Computer Performance Modelling, Measurement, and Evaluation*, May 1983, 251-268.

[Hailpern 82] Hailpern, B. and Korth, H., "An Experimental Distributed Database System," IBM Research Report RC 9678, Yorktown, November 1982.

[Hammer 80] Hammer, M. and Shipman, D., "Reliability Mechanisms for SDD-1: A System for Distributed Databases," *ACM Transactions on Database Systems 5*:4, December 1980, 431-466.

[Harter 85] Harter, P., Heimbinger, D. and King, R., "Idd: An Interactive Distributed Debugger," *Proceedings of the 5th International Conference on Distributed Computing Systems*, May 1985, 498-506.

[Hatch 85] Hatch, M., Katz, M. and Rees, J., "AT&T's RFS and Sun's NFS: A Comparison of Heterogeneous Distributed File Systems," *UNIX/World* 2:11, December 1985, 39-52.

[Herlihy 84] Herlihy, M., "Replication Methods for Abstract Data Types," MIT/LCS/TR-319, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1984.

[Howard 88] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R. and West, M., "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems 6*:1, February 1988, 51-81.

[Hu 86] Hu, I., "Measuring File Access Patterns in UNIX," *Performance Evaluation Review*, December 1986, 15-20.

[Jajodia 87] Jajodia, S. and Mutchler, D., "Enhancements to the Voting Algorithm," *Proceedings of the 13th VLDB Conference*, 1987, 399-406.

[Jenny 82] Jenny, C., "Placing Files and Resources in Distributed Systems: A General Method Considering Resources with Limited Capability," *IBM Research Report RZ 1157*, Zurich, July 1982.

[Joy 83] Joy, W., Cooper, E., Fabry, R., Leffler, S., McKusick, K. and Mosher, D., "4.2BSD System Manual," in *The UNIX Programmer's Manual, Seventh Edition, Virtual VAX-11 Version*, vol. 2, Bell Laboratories, modified by the University of California, Berkeley, California, March 1983.

[Jul 88] Jul, E., Levy, H., Hutchinson, N. and Black, A., "Fine-Grained Mobility in the Emerald System," *ACM Transactions on Computer Systems* 6:1, February 1988, 109-133.

[Kernighan 78] Kernighan, B. and Ritchie, D., *The C Programming Language*, Prentice-Hall, 1978.

[Kleinman 86] Kleinman, S., "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Proceedings of the 1986 Summer USENIX Conference*, June 1986, 238-247.

[Kohler 83] Kohler, W. and Jeng, B., "CARAT: A Distributed Software Testbed," TR CS-84-113, Electrical and Computer Engineering, University of Massachusetts at Amherst, December 1983.

[Lampson 80] Lampson, B., "Atomic Transactions," in *Lecture Notes for Advanced Course on Distributed Systems - Architecture and Implementation*, Institut fur Informatic Technische, Universitat Munchen, Munich, Germany, March 1980.

[Lantz 82] Lantz, K., Gradischnig, K., Feldman, J. and Rashid, R., "Rochester's Intelligent Gateway," *Computer 15*:10, October 1982, 54-70.

[Lantz 86] Lantz, K., Edighoffer, J. and Hitson, B., "Towards a Universal Directory Service," *Operating Systems Review 20*:2, April 1986, 43-53.

[Lawrie 82] Lawrie, D., Randal, J. and Barton, R., "Experiments with Automatic File Migration," *Computer 15*:7, July 1982, 45-56.

[Lazowska 86] Lazowska, E., Zahorjan, J., Cheriton, D. and Zwaenepoel, W., "File Access Performance of Diskless Workstations," *ACM Transactions on Computer Systems* 4:3, August 1986, 238-268.

[LeBlanc 84] LeBlanc, T., Gerber, R. and Cook, R., "The StarMod Distributed Programming Kernel," *Software - Practice and Experience 14*:12, December 1984, 1123-1140.

[Leffler 83] Leffler, S., Joy, W. and Fabry, R., "4.2BSD Networking Implementation Notes," CSRG TR/6, University of California, Berkeley, July 1983.

[Leffler 84] Leffler, S., Karels, M. and McKusick, M., "Measuring and Improving the Performance of 4.2BSD," *Proceedings of the 1984 USENIX Summer Conference*, June 1984, 237-252.

[Leffler 86] Leffler, S., Private Communication, August 1986.

[Lindsay 80] Lindsay, B. and Selinger, P., "Site Autonomy in R*: A Distributed Database Management System," IBM Research Report RJ 2927, San Jose, CA, September 1980.

[Lindsay 81] Lindsay, B., "Object Naming and Catalog Management for a Distributed Database Manager," *Proceedings of the 2nd International Conference on Distributed Computing Systems*, April 1981, 31-40.

[Lyon 85] Lyon, B., Sager, G., Chang, J., Goldberg, D., Kleinman, S., Lyon, T., Sandberg, R., Walsh, D. and Weiss, P., *Overview of the Sun Network File System*, Sun Microsystems, Inc., January 1985. (see also: *Networking on the Sun Workstation*, 800-1177-01, Sun Microsystems, Inc., May 1985.).

[Mayer 86] Mayer, J., Private communication, May 1986.

[McKusick 84] McKusick, M., Joy, W., Leffler, S. and Fabry, R., "A Fast File System For UNIX," *ACM Transactions on Computer Systems* 2:3, August 1984, 181-197.

[Metcalfe 76] Metcalfe, R. and Boggs, D., "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM* 19:7, July 1976, 395-404.

[Mitchell 79] Mitchell, J., Maybury, W. and Sweet, R., "Mesa Language Manual, Version 5.0," CSL-79-3, Systems Development Department, Xerox PARC, April 1979.

[Mitchell 82] Mitchell, J. and Dion, J., "A Comparison of Two Network-Based File Servers," *Communications of the ACM* 25:4, April 1982, 233-245.

[Mogul 86a] Mogul, J., Private Communication, July 1986.

[Mogul 86b] Mogul, J., "Representing Information about Files," TR STAN-CS-86-1103, Department of Computer Science, Stanford University, March 1986.

[Moore 82] Moore, L., Bukys, L. and Heliotis, J., "Design and Implementation of a Local Network Message Passing Protocol," *Proceedings of the 7th Conference on Local Computer Networks*, October 1982, 70-74.

[Morris 86] Morris, J., Satyanarayanan, M., Conner, M., Howard, J., Rosenthal, D. and Smith, F., "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM* 29:3, March 1986, 184-201.

[Moss 81] Moss, J. E., "Nested Transactions: An Approach to Reliable Distributed Computing," MIT/LCS/TR-260, Laboratory for Computer Science, Massachusetts Institute of Technology, April 1981.

[Mullender 84] Mullender, S. and Tanenbaum, A., "Immediate Files," *Software—Practice & Experience* 14:4, April 1984, 365-368.

[Murthy 83] Murthy, K., Kam, J. and Krishnamoorthy, M., "An Approximation Algorithm to the File Allocation Problem in Computer Networks," *2nd Symposium on Principles of Database Systems*, March 1983, 258-266.

[Nelson 88] Nelson, M., Welch, B. and Ousterhout, J., "Caching in the Sprite Network File System," *ACM Transactions on Computer Systems 6*:1, February 1988, 134-154.

[Noe 86] Noe, J. and Andreassian, A., "Effectiveness of Replication in Distributed Computer Networks," TR 86-06-05, Department of Computer Science, University of Washington, July 1986.

[Notkin 87] Notkin, D., Hutchinson, N., Sanislo, J. and Schwartz, M., "Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity," *Communications of the ACM 30*:2, February 1987, 132-140.

[Nowitz 78] Nowitz, D. and Lesk, M., "A Dial-Up Network of UNIX Systems," in *The UNIX Programmer's Manual, Seventh Edition*, vol. 2, Bell Laboratories, August 1978.

[Oppen 81] Oppen, D. and Dalal, Y., "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment," TR OPD-T8103, Xerox Office Products Division, October 1981.

[Ousterhout 85] Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M. and Thompson, J., "A Trace Driven Analysis of the UNIX 4.2BSD File System," *Operating Systems Review 19*:5, December 1985, 15-24.

[Ousterhout 88] Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M. and Welch, B., "The Sprite Network Operating System," *Computer 21*:2, February 1988, 23-36.

[Parker 82] Parker, D. and Ramos, R., "A Distributed File System Architecture Supporting High Availability," *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, 1982, 161-183.

[Porcar 82] Porcar, J., "File Migration in Distributed Computer Systems," TR LBL-14763, Lawrence Berkeley Laboratory, July 1982.

[Postel 82] Postel, J., "File Transfer Protocol," ARPANET RFC-765, Internet Protocol Transition Workbook, Network Information Center, SRI International, March 1982.

[Pu 86] Pu, C., Noe, J. and Proudfoot, A., "Regeneration of Replicated Objects: A Technique and Its Eden Implementation," *Proceedings of the International Conference on Data Engineering*, February 1986, 175-187.

[Quarterman 85] Quarterman, J., Silberschatz, A. and Peterson, J., "4.2BSD and 4.3BSD as Examples of the UNIX System," *ACM Computing Surveys 17*:4, December 1985, 379-418.

[Rashid 80] Rashid, R., "An Inter-Process Communication Facility for Unix," TR CMU-CS-80-124, Department of Computer Science, Carnegie-Mellon University, March 1980.

[Rashid 81] Rashid, R. and Robertson, G., "Accent: A Communication Oriented Network Operating System Kernel," *Operating Systems Review 15*:5, December 1981, 64-75.

[Rifkin 86] Rifkin, A., Forbes, M., Hamilton, R., Sabrio, M., Shah, S. and Yueh, K., "RFS Architectural Overview," *Proceedings of the 1986 Summer USENIX Conference*, June 1986, 248-259.

[Ritchie 78] Ritchie, D. and Thompson, K., "The UNIX Time-Sharing System," *Bell System Technical Journal* 57:6, Part 2, July-August 1978, 1905-1930.

[Rowe 82] Rowe, L. and Birman, K., "A Local Network Based on the UNIX Operating System," *IEEE Transactions on Software Engineering SE-8*:2, March 1982, 137-146.

[Saltzer 79] Saltzer, J., "Naming and Binding of Objects," in *Operating Systems: An Advanced Course*, Springer-Verlag, 1979, 99-208.

[Sansom 86] Sansom, R., Julin, D. and Rashid, R., "Extending a Capability Based System into a Network Environment," TR CMU-CS-86-115, Computer Science Department, Carnegie-Mellon University, April 1986.

[Satyanarayanan 81] Satyanarayanan, M., "A Study of File Sizes and Functional Lifetimes," *Operating Systems Review 15*:5, December 1981, 96-108.

[Satyanarayanan 83] Satyanarayanan, M., "A Methodology for Modelling Storage Systems and its Application to a Network File System," TR CMU-CS-83-109, Department of Computer Science, Carnegie-Mellon University, March 1983.

[Satyanarayanan 85] Satyanarayanan, M., Howard, J., Nichols, D., Sidebotham, R., Spector, A. and West, M., "The ITC Distributed File System: Principles and Design," *Operating Systems Review 19*:5, December 1985, 35-50.

[Schantz 86] Schantz, R., Thomas, R. and Bono, G., "The Architecture of the Cronus Distributed Operating System," *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986, 250-259.

[Schroeder 84] Schroeder, M., Birrell, A. and Needham, R., "Experience with Grapevine: The Growth of a Distributed System," *ACM Transactions on Computer Systems* 2:1, February 1984, 3-23.

[Scott 85] Scott, M., "Design and Implementation of a Distributed Systems Language," TR 596, Computer Sciences Department, University of Wisconsin-Madison, May 1985.

[Shelzer 85] Shelzer, A., *Network Transparency Issues in an Internetwork Environment*, Ph.D. Dissertation, Computer Science Department, UCLA, 1985.

[Shelzer 86] Shelzer, A., Lindell, R. and Popek, G., "Name Service Locality and Cache Design in a Distributed Operating System," *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986, 515-522.

[Sheng 86] Sheng, O. R. L., *Models for Dynamic File Migration in Distributed Computer Systems*, Ph.D. Dissertation, Graduate School of Management, University of Rochester, November 1986.

[Skeen 81] Skeen, D., "A Decentralized Termination Protocol," *Proceedings of the Symposium on Reliability in Distributed Software and Systems*, July 1981, 27-32.

[Smith 81a] Smith, A., "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms," *IEEE Transactions on Software Engineering SE-7*:4, July 1981, 403-417.

[Smith 81b] Smith, A., "Long Term File Migration: Development and Evaluation of Algorithms," *Communications of the ACM 24*:8, August 1981, 521-532.

[Smith 81c] Smith, E., "Debugging Techniques for Communicating, Loosely Coupled Processes," TR 100, Department of Computer Science, University of Rochester, December 1981.

[Smith 84] Smith, W. and Decitre, P., "An Evaluation Method for Analysis of the Weighted Voting Algorithm for Maintaining Replicated Data," *Proceedings of the Fourth International Conference on Distributed Computing Systems*, May 1984, 494-502.

[Smith 85] Smith, A., "Disk Cache-Miss Ratio Analysis and Design Considerations," *ACM Transactions on Computer Systems 3*:3, August 1985, 161-203.

[Stonebraker 79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Transactions on Software Engineering SE-5*:3, 1979, 188-194.

[Stritter 77] Stritter, E., "File Migration," TR STAN-CS-77-592, Stanford University, March 1977.

[Stroustrup 84] Stroustrup, B., "A Set of C++ Classes for Co-Routine Style Programming," Computer Science Technical Report 90, AT&T Bell Laboratories, November 1984.

[Sturgis 80] Sturgis, H., Mitchell, J. and Israel, J., "Issues in the Design and Use of a Distributed File System," *Operating Systems Review 14*:3, July 1980, 55-69.

[Svobodova 84] Svobodova, L., "File Servers for Network-Based Distributed Systems," *ACM Computing Surveys 16*:4, December 1984, 353-398.

[Svobodova 85] Svobodova, L., "Workshop Summary - Operating Systems in Computer Networks," *Operating Systems Review 19*:2, April 1985, 6-39.

[Terry 86] Terry, D., "Structure-free Name Management for Evolving Distributed Environments," *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986, 502-508.

[Terry 87] Terry, D. B., "Caching Hints in Distributed Systems," *IEEE Transactions on Software Engineering SE-13*:1, January 1987, 48-54.

[Thacker 79] Thacker, C., McCreight, E., Lampson, B., Sproull, R. and Boggs, D., "Alto: A Personal Computer," TR CSL-79-11, Xerox Palo Alto Research Center, August 1979.

[Tichy 84] Tichy, W. and Zuwang, R., "Towards a Distributed File System," *Proceedings of the 1984 USENIX Summer Conference*, June 1984, 87-97.

[Wah 80] Wah, B., "An Efficient Heuristic for File Placement on Distributed Databases," *COMPSAC 80*, October 1980, 462-468.

[Walker 83] Walker, B. J., *Issues of Network Transparency and File Replication in Distributed Systems, LOCUS*, Ph.D. Dissertation, Computer Science Department, University of California, Los

Angeles, 1983.

[Walker 83b] Walker, B., Popek, G., English, R., Kline, C. and Thiel, G., "The LOCUS Distributed Operating System," *Operating Systems Review 17*:5, December 1983, 49-70.

[Watson 81] Watson, R., "Identifiers (Naming) in Distributed Systems," in *Distributed Systems - Architecture and Implementation: An Advanced Course*, Springer-Verlag, 1981, 191-210.

[Xerox 79] *Mesa System Documentation, Version 5.0*, Xerox PARC, Systems Development Department, April 1979.

[Young 87] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D. and Baron, R., "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Operating Systems Review 21*:5, November 1987, 63-76.

[Zhou 85] Zhou, S., Da Costa, H. and Smith, A., "A File System Tracing Package for Berkeley UNIX," TR UCB/CSD 85/235, EECS Department, University of California, Berkeley, May 1985.